
documentation Plone5 Documentation

Release latest

May 17, 2017

Contents

1	Introduction	3
1.1	What is Plone?	3
1.2	What does Plone mean? How is it pronounced?	4
2	Quickstart	5
2.1	Online demo sites	5
2.2	Plone on your own machine	5
2.3	Deployment	6
3	Working with Content	7
3.1	Introduction	7
3.2	Logging In	24
3.3	Adding Content	26
3.4	Managing Content	59
3.5	Using TinyMCE as visual editor	125
3.6	Collaboration and Workflow	134
3.7	Using Listings & Queries (Collections)	144
3.8	Portlet Management	157
3.9	Create and maintain good quality content	164
4	Adapting & Extending Plone	167
4.1	Basic Changes (Look and Feel)	167
4.2	Theming Plone	172
4.3	Site Setup	220
4.4	Installing Add-ons	263
4.5	Custom Content Types	263
5	Installing, Managing and Updating Plone	265
5.1	Installing Plone	265
5.2	Guide to deploying and installing Plone in production	282
5.3	Plone Upgrade Guide	377
5.4	Troubleshooting	399
5.5	Automating Plone Deployment	467
6	Developing for Plone	469
6.1	Develop Plone Add ons	469
6.2	Programming with Plone	577

6.3	Debugging Plone	1047
6.4	Testing Plone	1055
6.5	Styleguides	1071
6.6	Importing content from other sources	1088
6.7	Tutorials	1090
6.8	Selected Plone core package documentation	1090
7	Appendices	1091
7.1	Glossary	1091
7.2	Error Reference	1097
7.3	older manuals	1097
8	About this documentation	1099
8.1	Writing Documentation	1099
8.2	Documentation Styleguide	1103
8.3	ReStructuredText Style Guide	1107
8.4	Word choice	1113
8.5	Helper tools for writing Documentation	1113
8.6	Searching the documentation	1115
9	License for Plone Documentation	1117
10	Asking for help	1119
10.1	Guidelines and Examples	1119

This is a community-maintained manual for the [Plone](#) content management system.

This documentation is for:

- Content editors: who write, update, and order content on a site
- Site administrators: who install Plone and add-ons, and set up a site
- Designers: who create site themes
- Deployers: who configure server(s) for site hosting
- Developers: who customize a site's capabilities, create add-ons, and contribute to Plone itself

What is Plone?

A powerful, flexible Content Management Solution that is easy to install, use and extend.

Plone lets non-technical people create and maintain information for a public website or an intranet using only a web browser. Plone is easy to understand and use - allowing users to be productive in just half an hour - yet offers a wealth of community-developed add-ons and extensibility to keep meeting your needs for years to come.

Blending the creativity and speed of open source with a technologically advanced [Python](#) back-end, Plone offers superior security without sacrificing power or extensibility.

The Plone community is an incredibly diverse group that bridges many types and sizes of organizations, many countries and languages, and everything from technical novices to hardcore programmers. Out of that diversity comes an attention to detail in code, function, user interface and ease of use that makes Plone one of the top 2% of open source projects worldwide. (Source: [Ohloh](#))

Plone's intellectual property and trademarks are protected by the non-profit [Plone Foundation](#). This means that Plone's future is not in the hands of any one person or company.

Thousands of websites across large and small businesses, education, government, non-profits, sciences, media and publishing are using Plone to solve their content management needs. This is supported by a global network of [over 300 solution providers](#) in more than 50 countries. Looking for a hosting provider to host your Plone site for you? You can find a list of providers and consultants on [plone.org](#).

We are very proud to be known by the company we keep. Organizations as diverse as NASA, Oxfam, Amnesty International, Nokia, eBay, Novell, the State Universities of Pennsylvania and Utah, as well as the Brazilian and New Zealand governments - all use Plone.

Plone is open on many levels. It runs on Linux, Windows, Mac OS X, FreeBSD and Solaris - and offers a straightforward installation to get you up and running in minutes. It has been translated into more than 40 languages, and is developed with an unflinching emphasis on usability and standards compliance.

Need a CMS that integrates with Active Directory, Salesforce, LDAP, SQL, Web Services. LDAP or Oracle? Plone does. Need to be sure your website is accessible? Plone meets or exceeds US Government 508 and the WCAG 2.0 standards.

Worried about security? As an open source product, a large number of developers frequently scrutinize the code for any potential security issues. This proactive approach is better than the wait-and-see approach in proprietary software that relies on keeping security issues a secret instead of resolving them outright. Based on Python and the Zope libraries, Plone has a technological edge that has helped it attain the best security track record of any major CMS (Source: [CVE](#)). In fact, security is a major reason why many CMS users are switching to Plone.

The market is full of open source content management systems, so it is important to do your homework before choosing one for your organization. Remember that a simple CMS may work out great to start with, but lead to problems with scaling or migration when you need more capability than it can provide. At the other end of the spectrum, a powerful CMS can be so difficult to learn and maintain that it never gains acceptance to users. Make sure the CMS you choose meets your needs today without compromising future growth. We hope you'll take the time to learn more about Plone and what it can do for you. We've created this site specifically as a gateway to information on how to get started with Plone, as well as information on development, solutions providers, events, news, and the wealth of product add-ons created by the community.

What does Plone mean? How is it pronounced?

Does the word Plone mean anything, and where does it come from? Why is Plone called Plone?

The word Plone originally comes from the electronic band Plone that used to exist on the Warp record label. The music is playful and minimalistic.

The founders of Plone-the-Software (Alan Runyan and Alexander Limi) were listening to Plone-the-Band when they met (as well as during the initial coding/design of Plone) - and one of the original quotes floating around at the time was that "Plone should look and feel like the band sounds". Thus, a legend was born. ;)

Plone-the-Band [broke up in 2001](#), but Plone-the-Software lives on.

Plone is pronounced in the same manner as the word "grown". It is not spelled out when you say it, and is not an abbreviation for anything.

CHAPTER 2

Quickstart

Description

A quick overview to get you up and running with Plone.

Online demo sites

Online demo sites allow users to experience the look and feel of Plone as well as check out its ease of use and features. These sites let users log in using a variety of roles so they can see the difference between what users, editors, and administrators see when working in Plone. You can find a list of available online demo sites at the [Try Plone section](#) on [plone.com](#).

Another interesting approach to try out Plone is to install it in a cloud service. That way, you can set up your own demo (or even very-light-weight production) Plone, which makes it a good fit to have Plone with your choice of add-ons be tested by your department or other group. Here are two blog posts that show you how to do this: [Plone on a free-tier Heroku](#) by Nejc Zupan and [Installing Plone 5 on Cloud9 IDE](#) by David Bain.

Plone on your own machine

You can find the download options at [plone.org/download](#)

The recommended and best supported way to deploy Plone, both for laptops as well as servers, is the Unified Installer. It can provide you both with a single instance with developer tools installed, as well as with multiple failover clients running against a database server.

What this means is that it will scale from quick evaluation to development to deployment, which will make your experience easier.

The catch is that the Unified Installer works on Linux and Unix-like systems (including macOS and other BSD's), not Windows. For Windows, there is a currently no binary installer for Plone 5 available, though we anticipate to have

one in the future. In all honesty it must be said that this is not the ideal way to work with Plone if you want to *develop* with it. A large portion of the toolchain is not readily available on Windows environments.

There is a highly workable alternative: Using **virtual machines**. The latest release of Plone also comes as a Virtualbox / Vagrant image. This will install a fully-working Plone for you in a virtual machine, but integrated with the host so that all your favourite Windows editing and development tools work. If you want to develop with Plone on a Windows machine, that would be your easiest option.

For macOS users we also advise to use virtual machines for casual testing, and the installation of the Universal. The Unified Installer works just fine under macOS, but does require use of the terminal.

You can find all information on using these different options at the [Installation](#) chapter in the “Managing, Administration” section of these docs.

Or, head straight to plone.org/download to get started now!

Deployment

Any deployment of Plone as a real-world site will usually entail setting up some more software. In almost all cases, you will want to have a webserver like NGINX or Apache in front, and a cache like Varnish to optimize response times.

Depending on your needs, you might also want redundant, high-availability options like ha-proxy, and monitoring tools to keep an eye on things and notify you when trouble arises.

A good selection of these tools is described in the [Guide to deploying and installing Plone in production](#).

Alternative ways to deploy

The Unified Installer itself is based on [buildout](#). If you’re working with Plone a lot, it is a good idea to get familiar with this tool and the relation with other Python module management tools.

Buildout can be used in a variety of ways, and many people use it to tweak their own development-instances and/or deployment instances. See [starzel.buildout](#) for a rather maximalized example.

Note however, that your chances of [getting help](#) on setup questions in the Plone support channels (IRC, community.plone.org, mailinglists) increase when other people can reproduce your outcomes, which is most efficiently done with the Unified Installer.

Besides that, there are many people deploying Plone as part of other deployment tools, be they Ansible, Salt, Chef, Puppet or the like. If you are familiar with these, you will most likely find others in the Plone community that share your enthusiasm.

There is an Ansible playbook, maintained by the Plone community, that can completely provision a remote machine to run a full-stack, production-ready Plone server with all the bells and whistles.

Introduction

A Conceptual Overview of Plone

Conceptual Overview

An explanation of Plone as a content management system

What is Plone?

Plone is a content management system (CMS) which you can use to build a web site. With Plone, ordinary people can contribute content to a web site without the help of a computer geek. Plone runs over the Web, too.

You don't need to install any special software on your computer. The word *content* is meant to be general, because you can publish so many types of information, including:

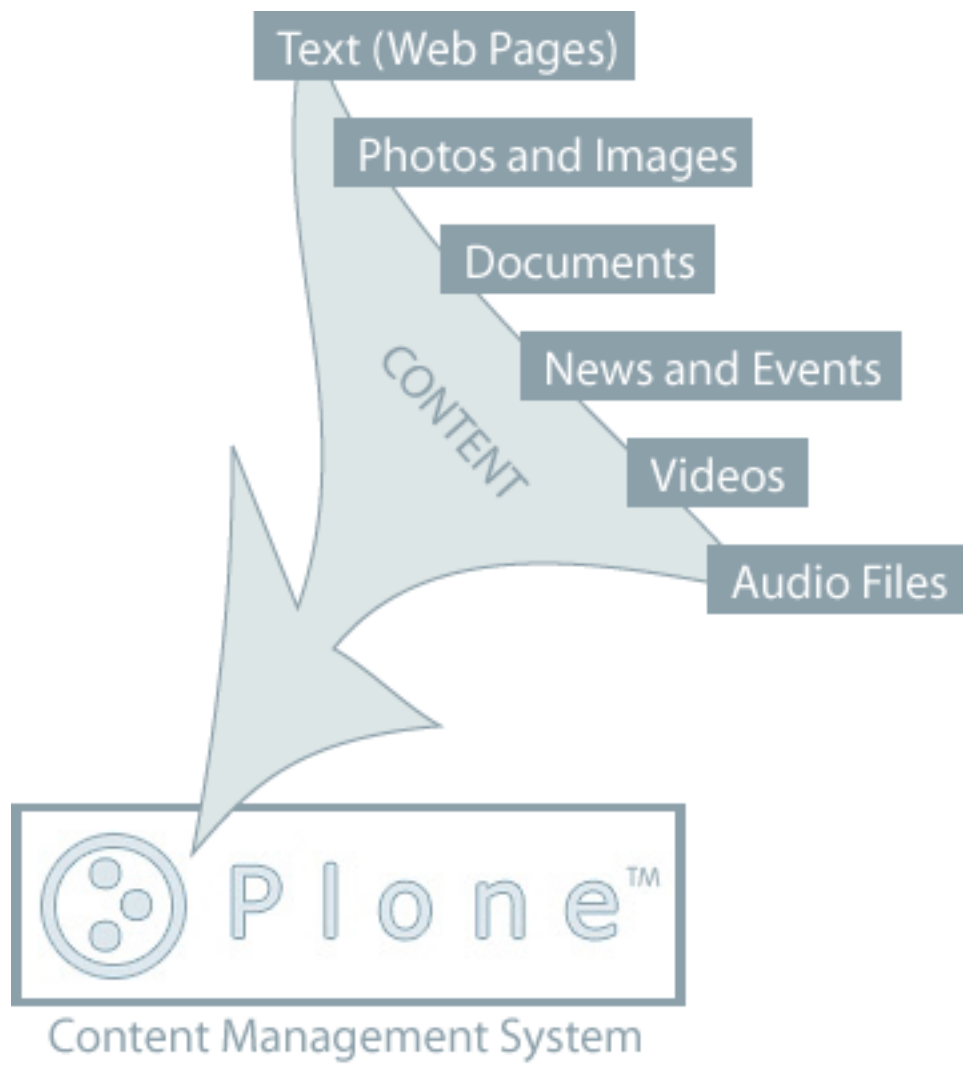
A Plone web site contains various kinds of content, including text, photos, and images. These can exist in many forms: documents, news items, events, videos, audio files, any types of file and data that can be uploaded or created on a web site. Content can also be uploaded from your local computer. You create *folders* on a Plone web site to hold content, and those automatically also create the navigation structure:

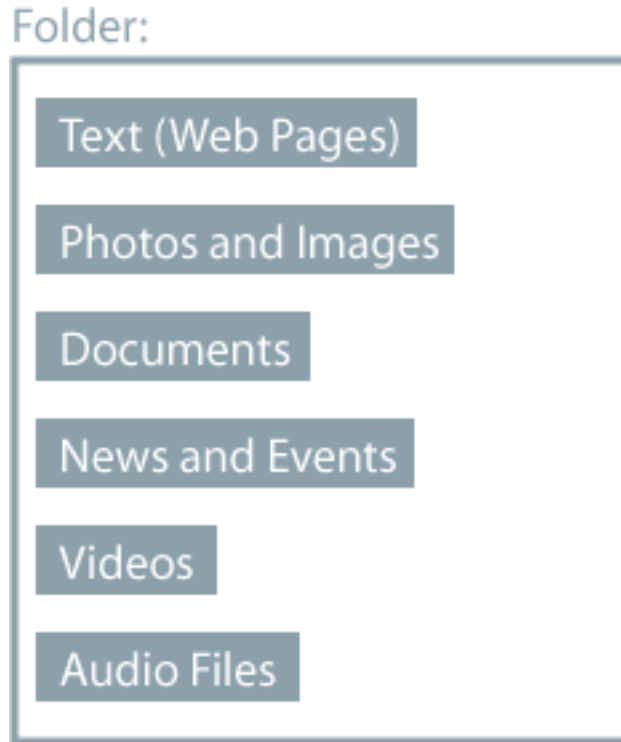
You Love Butterflies

For example, to add content about butterflies, you might add a folder named "Butterflies," then add some text to a web page in the folder:

And then you might add some butterfly photos to the folder:

You can add many types of content to a folder, including sub-folders. After adding a few reports and videos to the Butterflies folder, the content would be organized like this, with two sub-folders within the Butterflies folder:





What Goes on Behind the Scenes

You may wonder how it all works. A typical Plone web site exists as an installation of Plone software on a web server. The web server may be anywhere, often at a website server company within a “rack” of computers dedicated to the task:

The diagram shows the many cables that connect individual server computers to the Internet, across fast network connections. Your Plone site is software and database storage software installed on one of the individual server computers. As you type or click on your computer, data is sent up and down the networking cables and communication channels of the Internet to interact with your Plone software installation on the server.

Let’s simplify the diagram showing how you interact with Plone:

You use your web browser – Firefox, Safari, Internet Explorer, etc. – to view and edit your Plone web site, and the changes are stored by the Plone software into its database storage system.

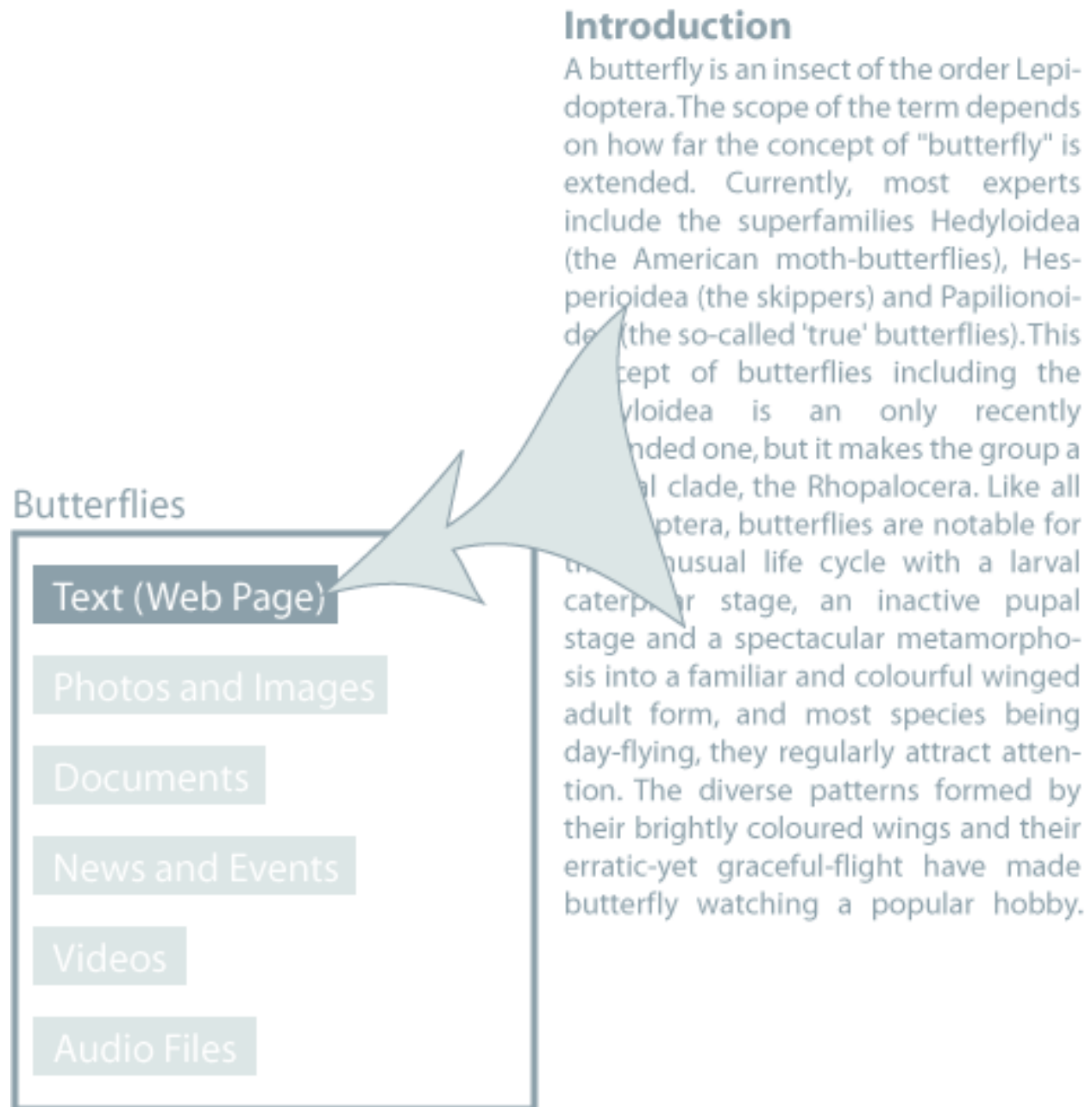
For example, imagine your butterfly Plone web site is located at `mysite.com`. You type `www.mysite.com` into your web browser. After you press Enter, the following sequence of events happens as your browser talks to the web server at `mysite.com`:

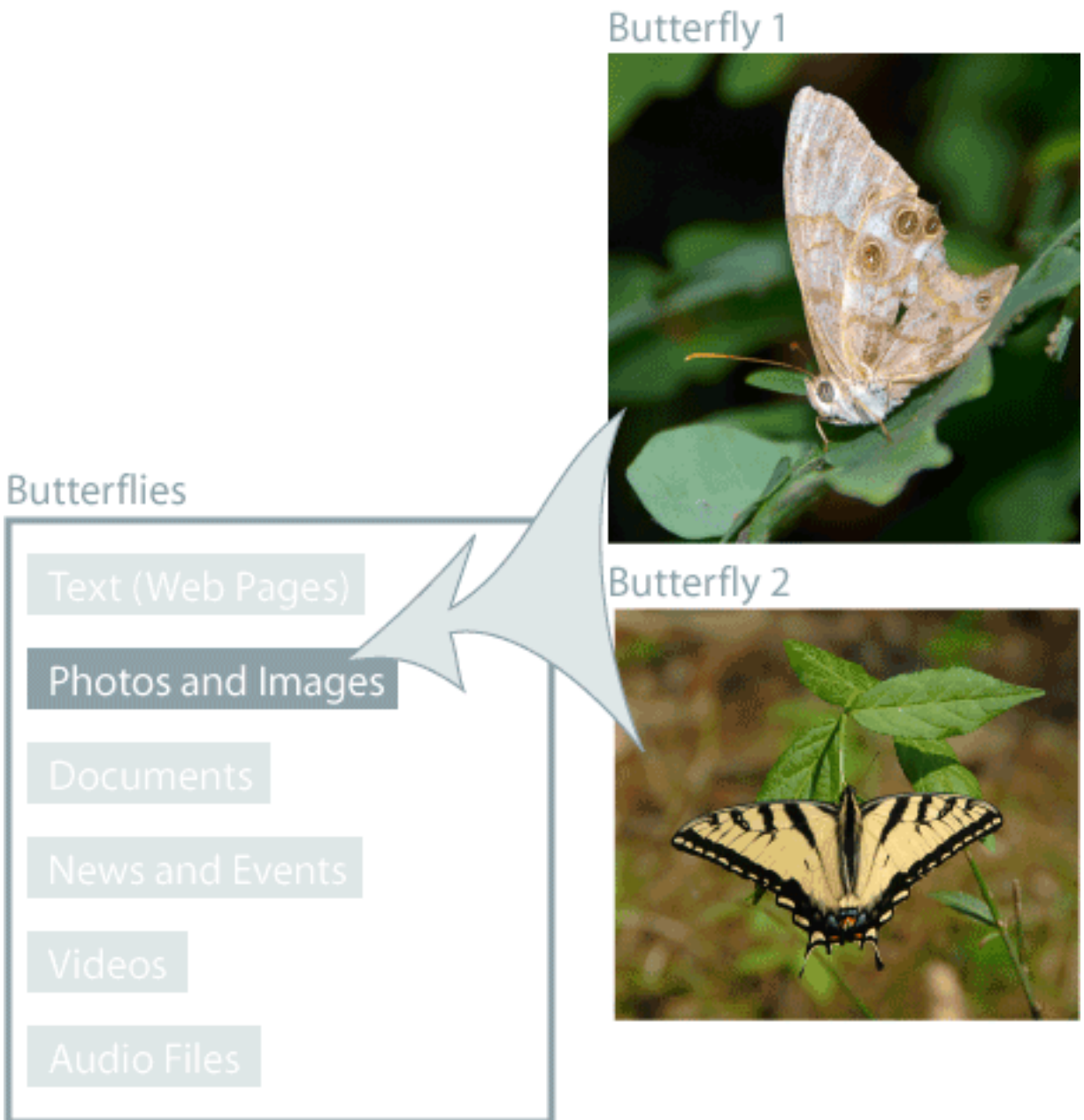
The Plone software responds:

Plone reads its database to look for information stored in `mysite.com`. It then sends back the web page to your computer, in a code called HTML. HTML is a computer language that describes how a web page looks. It includes text, graphics, fonts, the color of the background, and just about everything else. There are many online resources that can teach you HTML details, but one of Plone’s advantages is that you don’t need to know about HTML. That’s one reason for Plone and other similar web software; to let you focus on your content, e.g., butterfly text and graphics, instead of learning a new computer language.

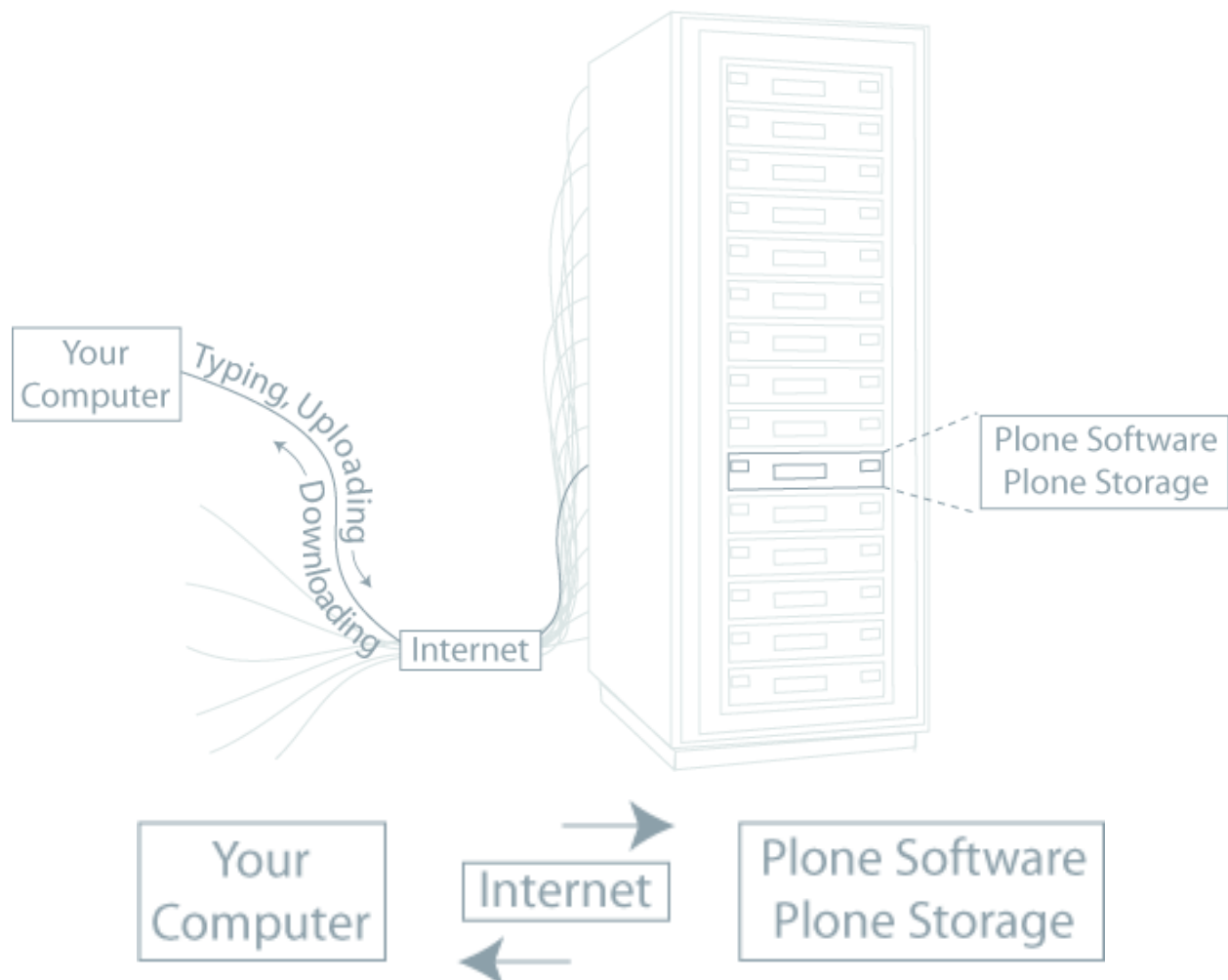
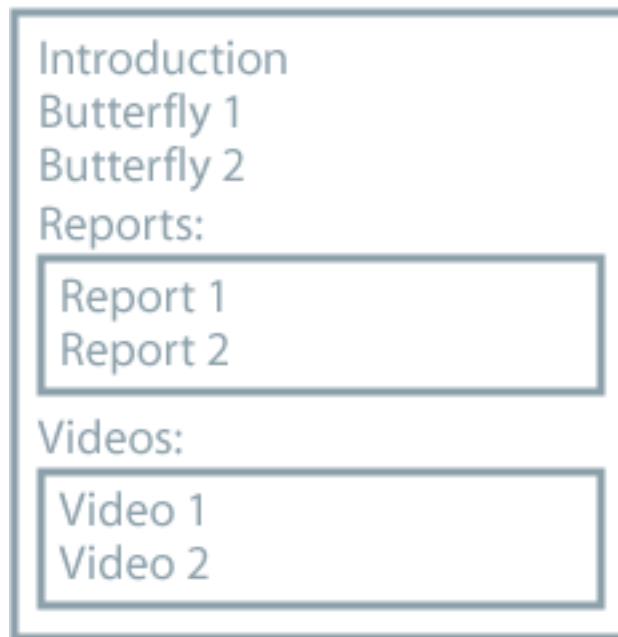
But back to our overview. Your web browser “renders” (translates) this HTML, and you see the resulting web page:

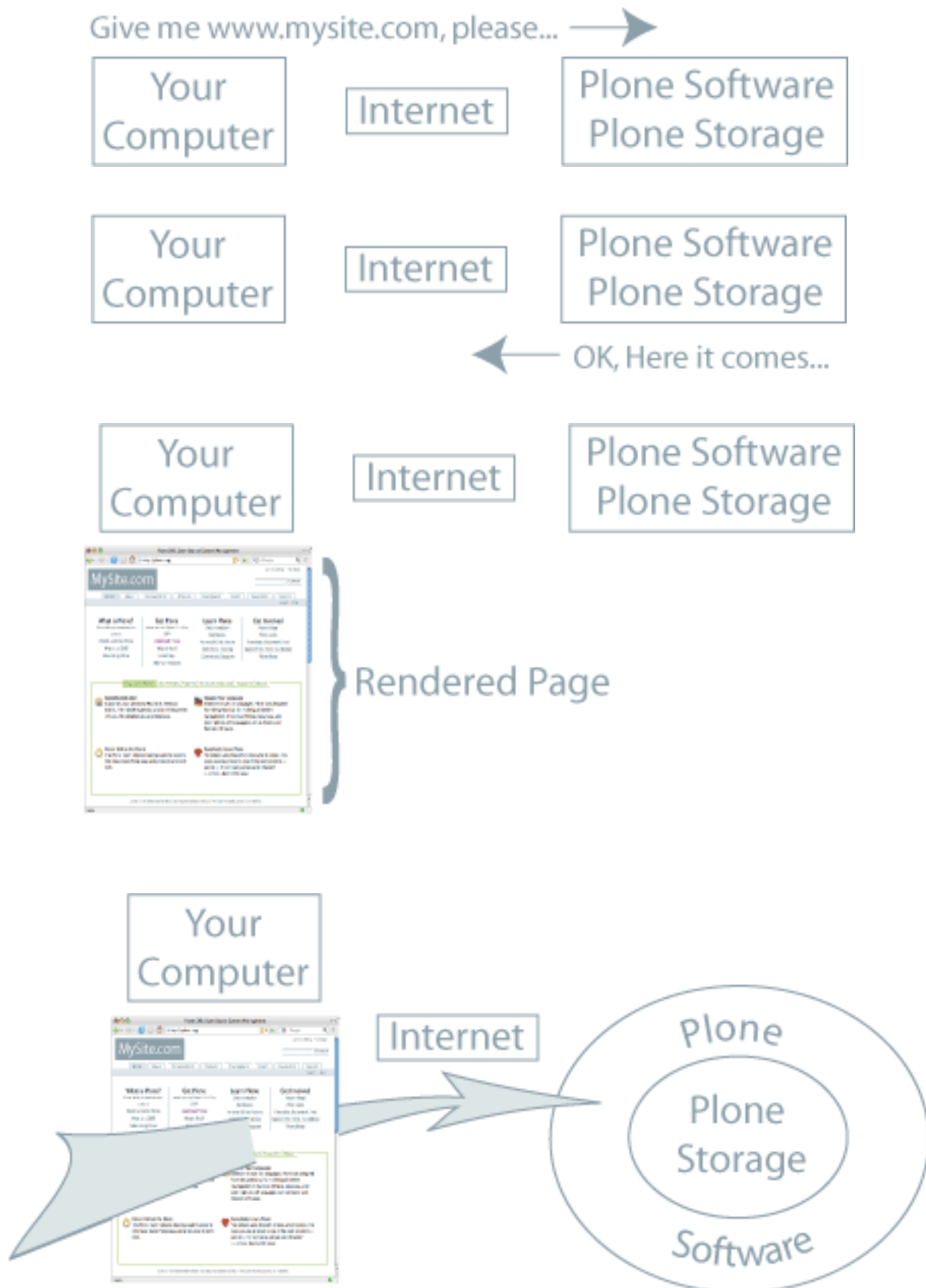
As you view your butterfly web page, you can choose to change it or add to it. You can also upload photos, documents, etc. at any time:



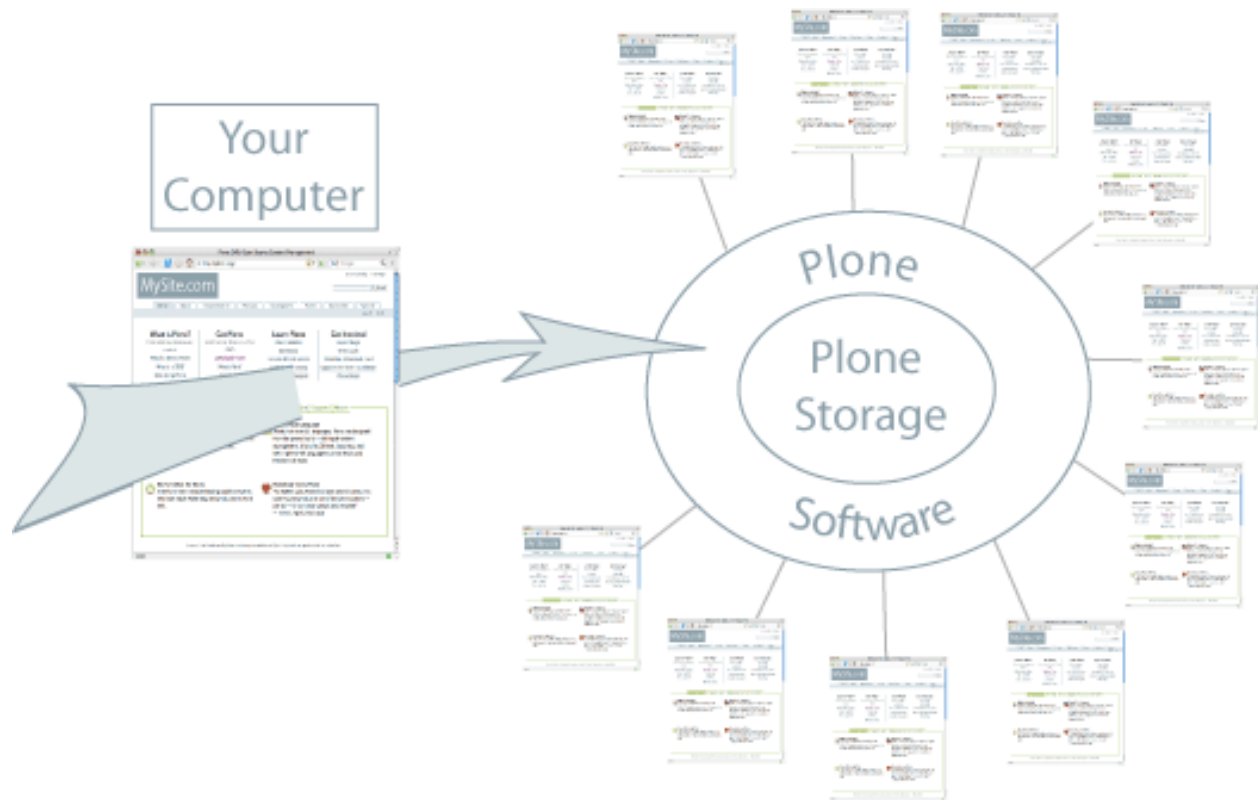


Butterflies:





After you make your edits and click “save changes,” the new version of the web page will be immediately available to anyone surfing to your site:



Visual Design Of Plone Web Sites

Plone allows web site administrators and designers the ability to create unique site designs. Here’s an overview of the Plone layout, and some design examples.

What does a Plone web site look like? For years there has been a consistent design for the default Plone appearance. The default design looks generally like this:

The Plone web site you use could have a design radically different from this, but you should be able to find common elements, such as the log in link and a navigation panel or menu. In the default design, the navigation menu is in the left area, and usually appears as an indented list of folders in the site. There also may be a set of tabs in the *Log In, Location Information* strip near the top.

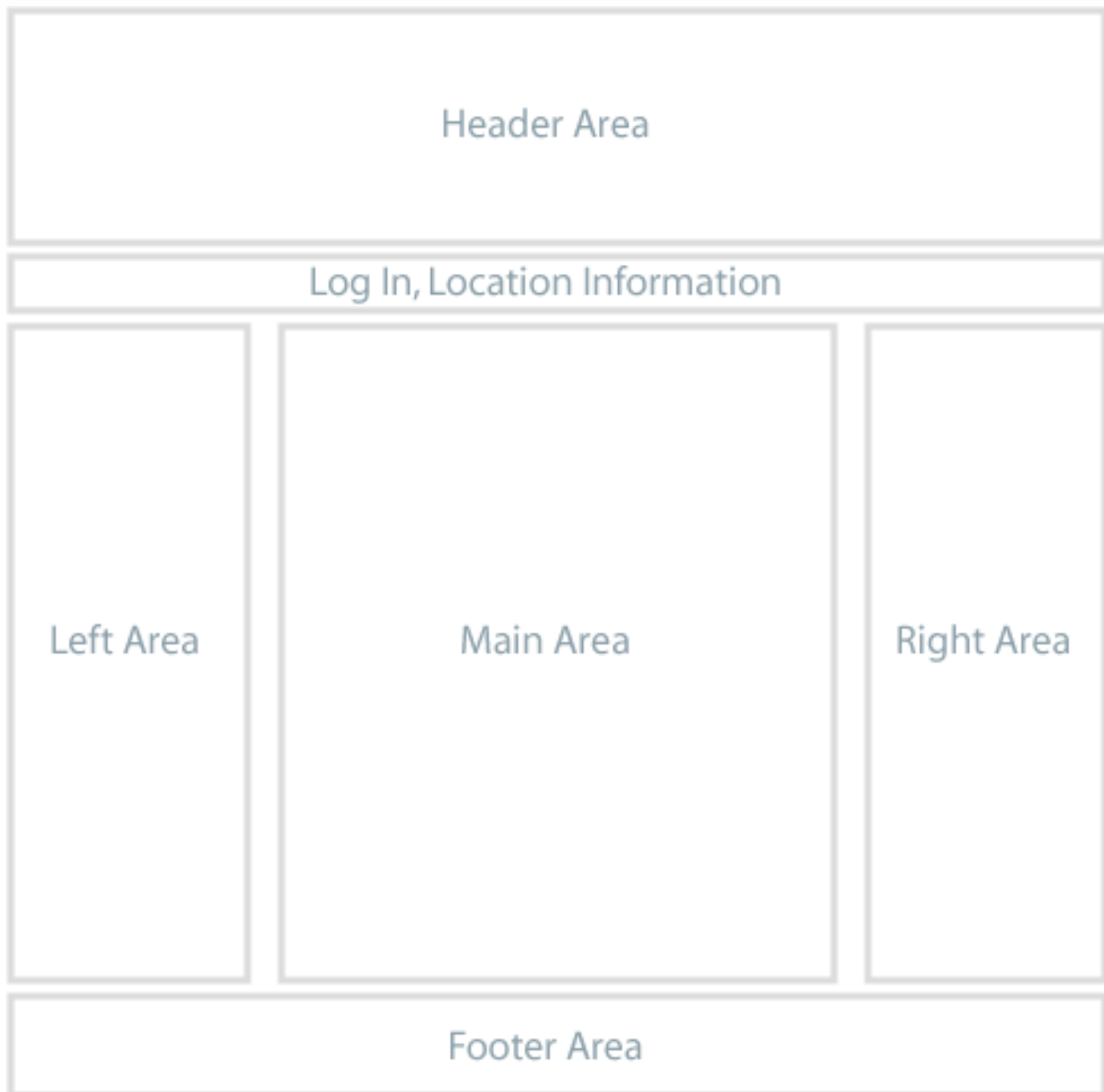
We can distinguish between the *design* of a web site and the *functionality* of a web site. Quite often, these aspects of a website are also under the control of different people with different skillsets. A designer will think about the layout, the appearance and the user interface, while a content editor will think about the structure of the information.

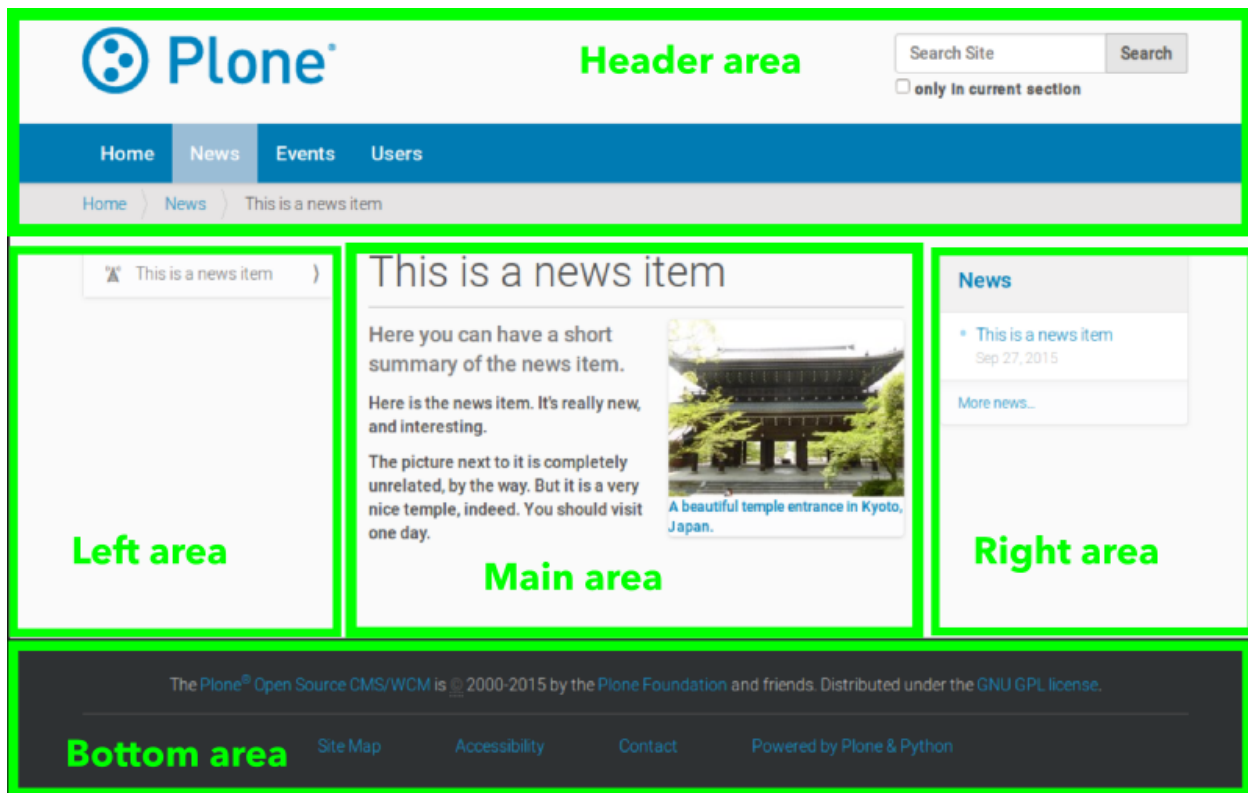
All of these aspects can be strictly separated in Plone, and can be adjusted independent from each other.

We’ll use the default Plone layout design as an example of typical divisions of the screen:

You may need to adapt these terms as needed for your Plone web site design. You may encounter varied terms for describing screen real estate, such as right and left “slots,” for the left and right column areas, “portlet,” or “viewlet,” for discrete areas or boxes, and several other terms.

For example, we can look at a web site from the [list of Plone success stories](#) to compare:





This is the Emergency Service web site for the Australian state of Victoria, giving citizens the latest update on potential natural disasters. The header section has the main navigation tabs, and a login for members. The left area lists the latest news. The main area has one large image and four columns of smaller images, serving as links to important areas of the website. At the bottom area (often known as “footer”) it has links to social media, legal information and contact information.

Nowadays, since the rise of mobile devices like tablets and smartphones, websites quite often look different depending on the screen size. On a smartphone, the navigation is often contracted into a single icon that expands when the user touches it. Also, the “left” and “right” areas may be shown after the “main” area when using a smartphone.

This technique is known as *responsive design*, and can be implemented using Plone. The default theme for Plone 5 uses this method already.

What does a Plone web site look like?

Traditionally, the out-of-the-box look is like that shown at the top of this page, with header, menu, columns, and a footer.

But using the flexible “Diazo” theme engine of Plone, each aspect can be made to look any way the designer chooses, and can also be shown different depending on the device of the visitor.

Plone User Accounts and Roles

```
*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot
```



Victoria State Emergency Service

FLOOD STORM
EMERGENCY

132 500



[Home](#)
[Warnings](#)
[About](#)
[Get Ready](#)
[Volunteer](#)
[EM Sector](#)
[Media](#)
[Support Us](#)
[Member Login](#)

Information in other languages:
한국어

Search Site

Search

Current Emergency Information



Everyone needs a Home Emergency Plan





Warnings

View current warnings and related community safety information.



Prepare

Learn how to prepare your home or business for emergencies.



Volunteer

Can you see yourself in orange?



Students

Primary, secondary and teacher resources and lots of fun stuff for kids

Latest News

A Chance encounter

Chance the horse is rescued by VICSES Manningham Unit

GOTAFE gets Emergency Safe

GOTAFE students find out about how to be FloodSafe and StormSafe

Get Ready for StormSafe Week

Do you know what to do to protect your property, possessions, livelihood and life during a storm?







[www.ses.vic.gov.au](#)
[Site Map](#)
[Accessibility](#)
[Contact](#)
[Copyright](#)
[Links](#)
[Disclaimer](#)
[Privacy](#)

3.1. Introduction

17

```
Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...            Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...            Open test browser
    Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
    Import library  Remote  ${PLONE_URL}/RobotRemote

    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...            Remote ZODB SetUp  ${FIXTURE}

    ${language} =  Get environment variable  LANGUAGE  'en'
    Set default language  ${language}

Test Teardown
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...            Remote ZODB TearDown  ${FIXTURE}

Suite Teardown
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...            Teardown Plone Site
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...            Close all browsers
```

Plone web sites come in many flavors, ranging from personal websites with one user to community, organization, or business websites that could have hundreds of users. Each person who adds content to a Plone web site has their own user account. A user account includes a user name and a password. Some Plone sites allow people to sign up by visiting the site, clicking a **Join** link, and filling out basic user information. Other sites have user accounts that are created by web site administrators, in which case people normally receive emails with the user account details.

However created, a Plone user account allows a person to log in by typing their user name and password. Passwords are case-sensitive, which means that you have to pay attention to the uppercase and lowercase letters. For example, if your password is xcFGt6v, you would have to type that exactly for it to work. Passwords that have a variety of characters in them are preferred over passwords like “raccoon” or “boardwalk,” as they are more difficult to guess and therefore more secure.

Anonymous vs Authenticated Web Surfing

The distinction between *anonymous web surfing* and *authenticated (logged-in) web activity* is an important one:

Anonymous Web Surfing

This is the normal experience for a person surfing the web. You type the web address of a web site into your browser and you look at web pages, watch videos, view images, but you don't have to log in. This is why this mode is called anonymous: anyone can do it just by surfing normally.

Note: Pro tip: you can use two different browsers (like Firefox and Chrome), and *not* log in with one of them.

That way you can compare how visitors will see your site, and you can spot content that is not yet published.

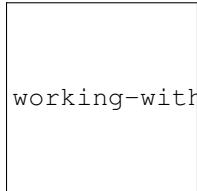
Note the presence of the *log in* link in the screen image below. If there is a *log in* link showing, you haven't logged in – and you are surfing the web site anonymously, as seen in the following screen capture of a new Plone web site:

```
*** Keywords ***

Highlight link
    [Arguments]  ${locator}
    Update element style  ${locator}  padding  0.5em
    Highlight  ${locator}

*** Test Cases ***

Take annotated screenshot
    Go to  ${PLONE_URL}
    Highlight link  css=#personaltools-login
    Capture and crop page screenshot
    ...  ${CURDIR}/../../_robot/anonymous-surfing.png
    ...  css=#content-header
    ...  css=#above-content-wrapper
```



working-with-content/introduction/../../_robot/anonymous-

Authenticated (Logged-in) Web Activity

You know the *authenticated* mode of web experience if you have ever used a bank or credit card website, or any other website that involved a user account. A bank web site will let you view your account information, fill out information forms, transfer funds, and do other actions, but only after you have logged in.

A Plone web site is not much different, except that you can do more sophisticated things.

Compare the screen image below, captured after a user called “Jane Doe” has logged in.

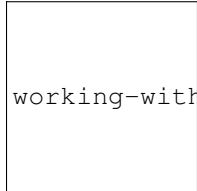
A toolbar has appeared on the left, which has a number of icons and actions that she can perform on the site content. At the bottom of the toolbar, her name is mentioned. A submenu opens when she clicks on this, allowing her to log out and perform various other options.

```
Enable autologin as  Manager
${user_id} = Translate  user_id
...  default=jane-doe
```

```
{user_fullname} = Translate user_fullname
... default=Jane Doe
Create user ${user_id} Member fullname=${user_fullname}
Set autologin username ${user_id}

*** Test Cases ***

Take logged in screenshot
Go to ${PLONE_URL}
Capture and crop page screenshot
... ${CURDIR}/../../../../_robot/loggedin-surfing.png
... css=#above-content-wrapper
... css=div.plone-toolbar-container
```

A rectangular box representing a screenshot of a Plone toolbar. The text 'working-with-content/introduction/../../../../_robot/loggedin-surfing.png' is written across the middle of the box, likely indicating the file path of the screenshot.

working-with-content/introduction/../../../../_robot/loggedin-surfing.png

User Roles

Equally important is the distinction between different user roles on a Plone web site. To illustrate the simplest case, let's consider two user roles, one called *member* and the other called *manager*. Consider the different rights or “power” of these two roles:

Member

- has a user account, so can log in
- can add content, but only in specific areas, and can't change anything outside of this area; often users are given a “home area,” to treat as personal space where they can add content.
- can not publish content so that it is visible to the anonymous web surfer, even content which they added; a person with manager role must approve content for publishing

Note: In many organizations, members are allowed to publish content on parts or all of the site. This is a policy that you can set up for each site, or even part of the site

Manager

- has a user account, so can log in
- can add content anywhere and has the power to change anything
- can publish any content

When you get your new account on a Plone web site, you should be given information indicating where you have the right to add content, after you have logged in.

After logging in, if you go to a folder where you have rights, you'll see the toolbar contain the appropriate options for *contents*, *view*, *edit* and *sharing* plus even more, depending on the role you have.

You'll be able to explore to find the differences between these options, but here are descriptions to help you start:

- *Contents* - shows a list of items in a folder. Also allows to re-arrange
- *Edit* - shows a page where you can edit the contents using a familiar editor
- *View* - shows the view an anonymous web surfer would see
- *Add new* - shows a sub-menu where you can edit new content items (images, pages, folders, etc.)
- *State* - shows menu choices for setting publication state (private, public draft, public, etc.)
- *Display* - shows menu choices for setting the display type (list view, summary view, etc.)
- *Sharing* - shows a panel for setting rights of other users to see or edit content

The toolbar will offer the main ways you interact with Plone. You will become very familiar with them as you learn more about managing a Plone-powered website.

Setting Your Preferences

```
*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Open test browser
    Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
    Import library  Remote  ${PLONE_URL}/RobotRemote

    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Remote ZODB Setup  ${FIXTURE}

    ${language} =  Get environment variable  LANGUAGE  'en'
    Set default language  ${language}

    Enable autologin as  Manager
    ${user_id} =  Translate  user_id
    ...  default=jane-doe
```

```
{user_fullname} = Translate user_fullname
... default=Jane Doe
Create user ${user_id} Member fullname=${user_fullname}
Set autologin username ${user_id}

Test Teardown
Run keyword if sys.argv[0].startswith('bin/robot')
... Remote ZODB TearDown ${FIXTURE}

Suite Teardown
Run keyword if not sys.argv[0].startswith('bin/robot')
... Teardown Plone Site
Run keyword if sys.argv[0].startswith('bin/robot')
... Close all browsers
```

After logging in to a Plone web site, you can change your personal preferences for information about your identity and choice of web site settings.

After logging in, your full name will show on the *toolbar*.

Click on your name to open the sub-menu, then click on the *Preferences* link to go to your personal area:

```
*** Test Cases ***

Show menubar
Go to ${PLONE_URL}

Click link css=#portal-personaltools a

Wait until element is visible
... css=#portal-personaltools li.plone-toolbar-submenu-header

Mouse over personaltools-preferences
Update element style portal-footer display none

Capture and crop page screenshot
... ${CURDIR}/../../../../_robot/show-preferences.png
... css=div.plone-toolbar-container
... css=li.plone-toolbar-submenu-header
```

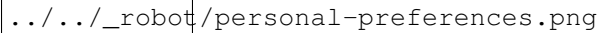


working-with-content/introduction/../../../../_robot/show-prefe

```
*** Test Cases ***

Show personal preferences
Go to ${PLONE_URL}/@@personal-preferences

Capture and crop page screenshot
... ${CURDIR}/../../../../_robot/personal-preferences.png
... css=#main-container
```



Date entry fields include:

- *Wysiwyg editor* - Plone comes standard with *TinyMCE*, an easy to use graphical editor to edit texts, link to other content items and so forth. Your site administrator might have installed alternatives, though.
- *Language* - On multilingual sites, you can select the language that you create content in most often. Plone excels at offering multilingual support.
- *Time zone* - If you work in a different timezone than the server default, you can select it here.

Personal information

Now let's switch over to the "Personal Information" tab:

```
*** Test Cases ***
```

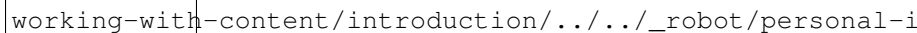
```
Show personal information
```

```
Go to  ${PLONE_URL}/@@personal-information
```

```
Capture and crop page screenshot
```

```
...  ${CURDIR}/../../_robot/personal-information.png
```

```
...  css=#main-container
```



- *Full Name*- If your name is common, include your middle initial or middle name.
- *E-mail address* - REQUIRED - You may receive emails from the website system, or from a message board, if installed, etc. When an item is required, a little red dot will show alongside the item.
- *Home page* web address - If you have your own web site or an area at a photo-sharing web site, for instance, enter the web address here, if you wish, so people can find out more about you.
- *Biography* text box - Enter a short description of yourself here, about a paragraph or so in length.
- *Location* - This is the name of your city, town, state, province, or whatever you wish to provide.
- *Portrait* photograph upload - The portrait photograph will appear as a small image or thumbnail-size image, so it is best to use a head shot or upper-torso shot for this.

You can change your preferences whenever you wish.

Changing your password

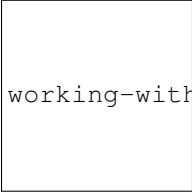
The last tab allows you to change your password.

Note: Plone is used by a variety of organisations. Some of these have centralized policies on where you can change your password, because this might also involve your access to other computer resources. In those cases, this screen might have been disabled.

```
*** Test Cases ***

Show personal information
    Go to  ${PLONE_URL}/@@change-password

    Capture and crop page screenshot
    ...  ${CURDIR}/../../../../_robot/change-password.png
    ...  css=#main-container
```



working-with-content/introduction/../../../../_robot/change-pas

Logging In

```
*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...            Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...            Open test browser
    Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
    Import library  Remote  ${PLONE_URL}/RobotRemote
```

```

Run keyword if sys.argv[0].startswith('bin/robot')
...           Remote ZODB SetUp  ${FIXTURE}

${language} = Get environment variable  LANGUAGE  'en'
Set default language  ${language}

Test Teardown
Run keyword if sys.argv[0].startswith('bin/robot')
...           Remote ZODB TearDown  ${FIXTURE}

Suite Teardown
Run keyword if not sys.argv[0].startswith('bin/robot')
...           Teardown Plone Site
Run keyword if sys.argv[0].startswith('bin/robot')
...           Close all browsers

```

What to expect when you log in to a Plone site

When you visit a Plone web site anonymously, or are given the web address for site maintenance, you'll see a *log in* button in the top-right corner like this:

```

*** Keywords ***

Highlight link
  [Arguments]  ${locator}
  Update element style  ${locator}  padding  0.5em
  Highlight  ${locator}

*** Test Cases ***

Take login link screenshot
  Go to  ${PLONE_URL}
  Highlight link  css=#personaltools-login
  Capture and crop page screenshot
  ...  ${CURDIR}/../_robot/login-link.png
  ...  css=#content-header
  ...  css=#above-content-wrapper

```

working-with-content/../../_robot/login-link.png

Note: Your site administrator may have *hidden* the login link, in which case you can go directly to a URL she or he has provided.

After clicking the *log in* link, you'll see an input panel where you can type in your user name and password:

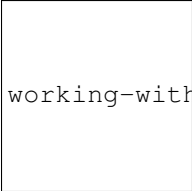
```

*** Test Cases ***

Take login screenshot
  Go to  ${PLONE_URL}/login
  Capture and crop page screenshot

```

```
...    ${CURDIR}/../_robot/login-popup.png
...    css=#content-core
```



working-with-content/../_robot/login-popup.png

After logging in to a Plone web site you will see some indication of your name, as the last item on the *toolbar*. You can click on your name to perform some actions related to your user, covered in the following sections.

You (or the site administrator) can allow users to use their email address as login name. This feature can be switched on in the Security settings control panel. The effect is that on the registration form no field is shown for the user name. On the login form the user is now asked to fill in an email address.

Adding Content

How to add basic content types to Plone web sites

Adding New Content

```
*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...            Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...            Open test browser
    Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
    Import library  Remote  ${PLONE_URL}/RobotRemote
```

```

Run keyword if sys.argv[0].startswith('bin/robot')
...           Remote ZODB SetUp  ${FIXTURE}

${language} = Get environment variable  LANGUAGE  'en'
Set default language  ${language}

Enable autologin as  Manager
${user_id} = Translate  user_id
... default=jane-doe
${user_fullname} = Translate  user_fullname
... default=Jane Doe
Create user  ${user_id}  Member  fullname=${user_fullname}
Set autologin username  ${user_id}

Test Teardown
Run keyword if sys.argv[0].startswith('bin/robot')
...           Remote ZODB TearDown  ${FIXTURE}

Suite Teardown
Run keyword if not sys.argv[0].startswith('bin/robot')
...           Teardown Plone Site
Run keyword if sys.argv[0].startswith('bin/robot')
...           Close all browsers

```

A general overview of how to add new content items in Plone, including definitions of each standard content type

New content items are added via the **Add New . . .** drop-down menu:

```

*** Test Cases ***

Show add new content menu
Go to  ${PLONE_URL}

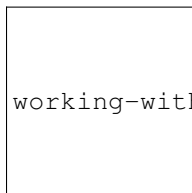
Wait until element is visible
...  css=span.icon-plone-contentmenu-factories
Click element  css=span.icon-plone-contentmenu-factories

Wait until element is visible
...  css=#plone-contentmenu-factories li.plone-toolbar-submenu-header

Mouse over  document
Update element style  portal-footer  display  none

Capture and crop page screenshot
...  ${CURDIR}/../../_robot/adding-content_add-menu.png
...  css=div.plone-toolbar-container
...  css=#plone-contentmenu-factories ul

```



working-with-content/adding-content/../../_robot/adding-c

Adding content in Plone is done *placefully*, which means you should navigate to the section of your Plone website where you want the new content to reside **before** you use the **Add New . . .** drop-down menu. You can of course cut,

copy, and paste content items from one section to another if needed at any later time.

Content Types

In Plone, you can use a number of **Content Types** to post certain kinds of content. For example, to upload an image you must use the **Image** content type. Below is a list of the available content types in order of their appearance, and what each are used for:

Collection Collections are used to group and display content based on a set of **criteria** which you can set. Collections work much like a query does in a database.


Event An Event is a content type specifically for posting information about an event (such as a fundraiser, meeting, barbecue, etc). This content type has a function which allows the site visitor to add the event to their desktop calendar. This includes applications such as: Google Calendar, Outlook, Sunbird and others. To add a single event to your calendar, click on the iCal link next to the “Add event to calendar” text in the main view of the event item.

Plone Conference 2015

The annual Plone worldwide gathering, this year in Bucharest, Romania.

This year's goals

- affirm the performance, innovation and team spirit surrounding the Plone community
- experience a fun and productive event that's packed with insightful lectures and unforgettable parties
- welcome Romania, a Plone Conf. debutant, and its Plone enthusiasts among the more established members

- **When**
Oct 12, 2015 to Oct 18, 2015
(Europe/Berlin / UTC200)
- **Where**
The Intercontinental Hotel, Bucharest
- **Attendees**
A wide variety of Plone enthusiasts
- **Web**
[Visit external website](#)
- **Add event to calendar**
 [iCal](#)

You can also get all the events in a folder in one go (currently only available in iCal format). To download the iCal file, append `@@ics_view` to the end of the URL of the folder or collection containing the events. For example, if you want to get all the events from the *events* folder in the root of your site, go to `http://example.com/events/@@ics_view`.

File A File in Plone is any binary file you wish to upload with the intent to be downloaded by your site visitors. Common examples are PDFs, Word Documents, and spreadsheets.

Folder Folders work in Plone much like they do on your computer. You can use folders to organize your content, and to give your Plone website a navigation structure.

Image The Image content type is used for uploading image files (JPG, GIF, PNG) so that you can insert them into pages or other page-like content types.

Link Also referred to as the ‘Link Object’; do not confuse this with the links you create with the visual editor on pages or other content types. The Link content type is often used to include a link to an external website in Navigation and other specialized uses.

News Item This content type is similar to a Page, only a News Item is specifically for posting news. You can also attach a thumbnail image to a News Item, which then appears in folder summary views next to the summary of the News Item.

Page A Page in Plone is the basic content types. Use Pages to write the bulk of your web pages on your Plone website.

Note: Depending on what add-on products you have installed, you may see more options in your **Add New . . .** drop-down menu than appear here. For information about those additional content types, refer to the Product documentation for the add-on in question.

Title

Nearly all content types in Plone have two fields in common: **Title** and **Description**.

The **Title** of content items, including folders, images, pages, etc., can be anything you want – you can use any keyboard characters, including spaces. **Titles** become part of web address for each item you create in Plone. Web addresses, also known as URLs, are what you type in a web browser to go to a specific location in a web site (Or, you would click your way there), such as:

www.mysite.com/about/personnel/sally/bio

or

www.mysite.com/images/butterflies/skippers/long-tailed-skippers

Web addresses *do* have restrictions on allowed keyboard characters, and spaces are not allowed. Plone does a good job of keeping web addresses correct by using near-equivalents of the **Title** that you provide, by converting them to lowercase, and by substituting dashes for spaces and other punctuation.

The web address of a given item is referred to as the **short name** in Plone. When you use the **Rename** function, you'll see the short name along with the title.

The fields will vary according to the content type. For instance, the Link content type has the URL field. The File content type has the File field, and so on.

Description

The **Description** appears at the top of pages, just under the Title. Descriptions are often used in conjunction with a variety of Folder and Collection views (such as Standard and Summary). The Description also appears in search results via Plone's native search engine.

The Description is just plain text, without any form of mark-up. This is to keep it inline with the *Dublin Core* standard, a long-established way of categorizing information.

Adding Folders

```
*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
```

```
@{APPLY_PROFILES} plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if not sys.argv[0].startswith('bin/robot')
    ...           Setup Plone site  ${FIXTURE}
    Run keyword if sys.argv[0].startswith('bin/robot')
    ...           Open test browser
    Run keyword and ignore error Set window size  @{DIMENSIONS}

Test Setup
    Import library Remote  ${PLONE_URL}/RobotRemote

    Run keyword if sys.argv[0].startswith('bin/robot')
    ...           Remote ZODB SetUp  ${FIXTURE}

    ${language} = Get environment variable LANGUAGE 'en'
    Set default language  ${language}

    Enable autologin as Manager
    ${user_id} = Translate user_id
    ... default=jane-doe
    ${user_fullname} = Translate user_fullname
    ... default=Jane Doe
    Create user  ${user_id} Member fullname=${user_fullname}
    Set autologin username  ${user_id}

Test Teardown
    Run keyword if sys.argv[0].startswith('bin/robot')
    ...           Remote ZODB TearDown  ${FIXTURE}

Suite Teardown
    Run keyword if not sys.argv[0].startswith('bin/robot')
    ...           Teardown Plone Site
    Run keyword if sys.argv[0].startswith('bin/robot')
    ...           Close all browsers
```

Adding folders to a Plone web site is the basic way of controlling the organization of content.

You have undoubtedly created folders (directories) on your computer's hard drive. Personal computers use a hierarchy of folders to structure and organize the programs and files on the hard drive. In Plone folders are essentially used the same way, except that they are created on a Plone web site, for organizing content in Plone's built-in storage system.

Folders are added by clicking the **Add new...** drop-down menu. Select **Folder** from the menu:

```
*** Test Cases ***

Show add new folder menu
    Go to  ${PLONE_URL}

    Wait until element is visible
    ... css=span.icon-plone-contentmenu-factories
    Click element  css=span.icon-plone-contentmenu-factories

    Wait until element is visible
    ... css=#plone-contentmenu-factories li.plone-toolbar-submenu-header
```

```

Mouse over  folder
Update element style  portal-footer  display  none

Capture and crop page screenshot
...  ${CURDIR}/../../../../_robot/adding-folders_add-menu.png
...  css=div.plone-toolbar-container
...  css=#plone-contentmenu-factories ul

```

working-with-content/adding-content/../../../../_robot/adding-f

You should now see the *Add Folder* screen:

```

*** Test Cases ***

Show new folder add form
    Page should contain element  folder
    Click link  folder

    Wait until element is visible
    ...  css=#form-widgets-IDublinCore-title

    Capture and crop page screenshot
    ...  ${CURDIR}/../../../../_robot/adding-folders_add-form.png
    ...  css=#content

```

working-with-content/adding-content/../../../../_robot/adding-f

Fill in the **Title**, which is required, as indicated by the red dot. The **Summary** is optional; you can always come back to the edit panel if you need to add a description of the folder. Summaries are useful when a site visitor uses the search tool included with Plone - results will display with both the Title and Summary of the item.

You also notice tabs along the top:

- *Default*, for entering the Title and Description fields,
- *Categorization*, for specifying categories that apply to the folder (you may know these as *keywords*),
- *Dates*, for setting the time period when the folder should be available for view on the web site,
- *Ownership*, for specifying the creator and/or contributors for the content item,
- *Settings*, for allowing comments about the item, enabling *Next/Previous Navigation*, and choosing whether it shows in the navigation menu for the web site.

These tabs are standard, so you'll see them when you click other content types. We will cover these tabs in another section of this user manual.

Be sure to click **Save** at the bottom of the page when you are finished. This will complete the folder creation process.

What's in a Web Name?

Individual content items on a Plone web site have discrete web addresses. Plone creates these automatically, based on the Title that you supply.

What's in a Web Name?

The **Title** of content items, including folders, images, pages, etc., can be anything you want – you can use any keyboard characters, including spaces. **Titles** become part of web address for each item you create in Plone. Web addresses, also known as URLs, are what you type in a web browser to go to a specific location in a web site (Or, you would click your way there), such as:

`www.mysite.com/about/personnel/sally/bio`

OR

`www.mysite.com/images/butterflies/skippers/long-tailed-skippers`

Web addresses *do* have restrictions on allowed keyboard characters, and spaces are not allowed. Plone does a good job of keeping web addresses correct by using near-equivalents of the **Title** that you provide, by converting them to lowercase, and by substituting dashes for spaces and other punctuation.

To illustrate, let's take each of these two web addresses and split them out into their component parts:

```
www.mysite.com/about/personnel/sally/bio
^
website name
      ^
      a folder named About
            ^
            a folder named Personnel
                  ^
                  a folder named Sally
                        ^
                        a folder named Bio
```

In this example, Plone changed each folder title to lowercase, e.g., from Personnel to personnel. You don't have to worry about this. Plone handles the web addressing; you just type in titles however you want.

And, for the second example:

```
www.mysite.com/images/butterflies/skippers/long-tailed-skippers
^
website name
      ^
      a folder named Images
            ^
            a folder named Butterflies
                  ^
                  a folder named Skippers
                        ^
                        a folder named Long-Tailed Skippers
```

This example is similar to the first, illustrating how there is a lowercase conversion from the title of each folder to the corresponding part of the web address. Note the case of the folder named Long-tailed Skippers. Plone kept the dash, as that is allowed in both title and part of the web address, but it changed the blank between the words Tailed and Skippers to a dash, in the web address, along with the lowercase conversion.

The web address of a given item is referred to as the **short name** in Plone. When you use the **Rename** function, you'll see the short name along with the title.

What's in a Title?

The title of a content item not only affects the **short name** that is used in the URL of the item. It is also displayed, with a twist, in the **title bar** of the browser window, or in the browser tab. The twist consists in the fact that what is displayed consists of the *item title* and the *site title*, separated by an **Em dash**. The site title is set in the *site control panel* (<http://yoursite.com/@@site-controlpanel>), but for the purposes of this section it is not necessary to have the permissions to access it.

For example, the title for the item at <https://www.cia.gov/about-cia> shows in the browser tab or title bar as:

About CIA — Central Intelligence Agency.

The part to the left of the Em dash, **About CIA** is the *item title*, while the part to the right, **Central Intelligence Agency**, is the *site title*. The site title is appended to the item title, with an Em dash, automatically. Technically, this is what the `<title>` HTML element is set to.

Why is this important? In and of itself, this behavior of a Plone site might often be overlooked. However, it becomes important when looking at the results provided by a search engine, such as Google. When Google lists a page from a Plone site, the title used is the same one just described (*item title — site title*).

Often, you might want the homepage of your site to be listed in Google search results with just the site title. But you can not leave the homepage item title empty, so how to achieve this? Thankfully, there is an easy solution: make the homepage title **exactly identical** to the site title.

In the CIA example above, if the homepage title were set to *Central Intelligence Agency*, then Google would list it simply as:

Central Intelligence Agency

Adding Images

```
*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Setup Plone site  ${FIXTURE}
```

```
Run keyword if sys.argv[0].startswith('bin/robot')
...           Open test browser
Run keyword and ignore error Set window size @{DIMENSIONS}

Test Setup
  Import library Remote ${PLONE_URL}/RobotRemote

  Run keyword if sys.argv[0].startswith('bin/robot')
  ...           Remote ZODB SetUp ${FIXTURE}

  ${language} = Get environment variable LANGUAGE 'en'
  Set default language ${language}

  Enable autologin as Manager
  ${user_id} = Translate user_id
  ... default=jane-doe
  ${user_fullname} = Translate user_fullname
  ... default=Jane Doe
  Create user ${user_id} Member fullname=${user_fullname}
  Set autologin username ${user_id}

Test Teardown
  Run keyword if sys.argv[0].startswith('bin/robot')
  ...           Remote ZODB TearDown ${FIXTURE}

Suite Teardown
  Run keyword if not sys.argv[0].startswith('bin/robot')
  ...           Teardown Plone Site
  Run keyword if sys.argv[0].startswith('bin/robot')
  ...           Close all browsers
```

Adding images to a Plone web site is a basic task that may involve a little work on your local computer, but is essential, because photographs, maps, and custom graphics are so important on web sites.

Preparing Images for the Web

Note: ****Remember to use web-standard file formats for all images. Acceptable formats include: JPG, JPEG, GIF, and PNG.**

Do not use BMP or TIFF formats as these are not widely supported by web browsers, and can lead to slower web-sites.**

When you are ready to upload an image, use the *Add new...* drop-down menu.

```
*** Test Cases ***

Show add new image menu
  Go to ${PLONE_URL}

  Wait until element is visible
  ... css=span.icon-plone-contentmenu-factories
  Click element css=span.icon-plone-contentmenu-factories

  Wait until element is visible
  ... css=#plone-contentmenu-factories li.plone-toolbar-submenu-header

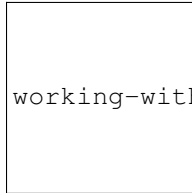
  Mouse over image
```

```

Update element style portal-footer display none

Capture and crop page screenshot
... ${CURDIR}/../../../../_robot/adding-images_add-menu.png
... css=div.plone-toolbar-container
... css=#plone-contentmenu-factories ul

```



working-with-content/adding-content/../../../../_robot/adding-i

After clicking to add an **Image**, you'll see the *Add Image* panel:

```

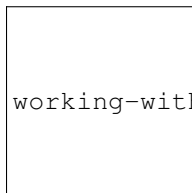
*** Test Cases ***

Show new image edit form
  Page should contain element image
  Click link image

  Wait until element is visible
  ... css=#form-widgets-title

  Capture and crop page screenshot
  ... ${CURDIR}/../../../../_robot/adding-images_add-form.png
  ... css=#content

```



working-with-content/adding-content/../../../../_robot/adding-i

The Title and Description fields (field, as in “data input field”) are there, as with adding a Folder, and at the bottom there is a place to upload an image. Let's look at the three input fields individually:

- *Title* - Use whatever text you want, even with blanks and punctuation (Plone handles web addressing).
- *Description* - Always a good idea, but always optional. Leave it blank if you want.
- *Image* - The Image field is a text entry box along with a Browse... button. You don't have to type anything here; just click the Browse button and you'll be able to browse you local computer for the image file to upload.

For images, at a minimum, you will browse your local computer for the image file, then click **Save** at the bottom to upload the image to the Plone web site.

You'll have to wait a few seconds for the upload to complete. A preview of the uploaded image will be shown when the upload has finished.

Images and files that you upload into Plone have their IDs (URLs) based on the title that you give to the image (instead of the file name of the image or file).

However, if you leave the title empty, the name of the item will default to the name of the file.

Adding Pages

```
*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Open test browser
    Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
    Import library  Remote  ${PLONE_URL}/RobotRemote

    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Remote ZODB SetUp  ${FIXTURE}

    ${language} =  Get environment variable  LANGUAGE  'en'
    Set default language  ${language}

    Enable autologin as  Manager
    ${user_id} =  Translate  user_id
    ...  default=jane-doe
    ${user_fullname} =  Translate  user_fullname
    ...  default=Jane Doe
    Create user  ${user_id}  Member  fullname=${user_fullname}
    Set autologin username  ${user_id}

Test Teardown
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Remote ZODB TearDown  ${FIXTURE}

Suite Teardown
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Teardown Plone Site
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Close all browsers
```

Pages in Plone vary greatly, but are single “web pages,” of one sort or another.

To add a page, use the *Add new...* menu for a folder:


```
*** Test Cases ***
```

```
Show add new page menu
```

```
Go to ${PLONE_URL}
```

```
Wait until element is visible
```

```
... css=span.icon-plone-contentmenu-factories
```

```
Click element css=span.icon-plone-contentmenu-factories
```

```
Wait until element is visible
```

```
... css=#plone-contentmenu-factories li.plone-toolbar-submenu-header
```

```
Mouse over document
```

```
Update element style portal-footer display none
```

```
Capture and crop page screenshot
```

```
... ${CURDIR}/../../../../_robot/adding-pages_add-menu.png
```

```
... css=div.plone-toolbar-container
```

```
... css=#plone-contentmenu-factories ul
```

working-with-content/adding-content/../../../../_robot/adding-p

Select **Page** from the menu, and you'll see the *Add Page* screen:

```
*** Test Cases ***
```

```
Show new page edit form
```

```
Page should contain element document
```

```
Click link document
```

```
Wait until element is visible
```

```
... css=#mceu_16-body
```

```
Capture and crop page screenshot
```

```
... ${CURDIR}/../../../../_robot/adding-pages_add-form.png
```

```
... css=#content
```

working-with-content/adding-content/../../../../_robot/adding-p

The **Title** and **Description** fields are there at the top. Fill each of them out appropriately. There is a *Change note* field at the bottom, also a standard input that is very useful for storing helpful memos describing changes to a document as you make them. This is useful for pages on which you may be collaborating with others.

The middle panel, **Body Text**, is where the action is for pages. The software used for making Pages in Plone, generically called *visual editor* and specifically a tool called TinyMCE, is a most important feature allowing you to do

WYSIWYG editing. WYSIWYG editing – *What You See Is What You Get* – describes how word processing software works. When you make a change, such as setting a word to bold, you see the bold text immediately.

People are naturally comfortable with the WYSIWYG approach of typical word processors. We will describe later in this manual.

Markup languages

Your site-administrator may also enable so-called markup languages. If you are the sort of person who likes to enter text using so-called mark-up formats, you may switch off the visual editor under your personal preferences, which will replace it with a simplified textentry panel. The mark-up formats available in Plone are:

- [Markdown](#)
- [Textile](#)
- [Structured Text](#)
- [Restructured Text](#)

Each of these works by the embedding of special formatting codes within text. For example, with structured text formatting, surrounding a word or phrase by double asterisks will make that word or phrase bold, as in ****This text would be bold.**** These mark-up formats are worth learning for speed of input if you do a lot of page creation, or if you are adept at such slightly more technical approaches to entering text. Some people prefer such formats not for speed itself, but for fluidity of expression.

Adding Files

```
*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Open test browser
    Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
    Import library  Remote  ${PLONE_URL}/RobotRemote
```

```

Run keyword if sys.argv[0].startswith('bin/robot')
...           Remote ZODB SetUp  ${FIXTURE}

${language} = Get environment variable  LANGUAGE  'en'
Set default language  ${language}

Enable autologin as  Manager
${user_id} = Translate  user_id
... default=jane-doe
${user_fullname} = Translate  user_fullname
... default=Jane Doe
Create user  ${user_id}  Member  fullname=${user_fullname}
Set autologin username  ${user_id}

Test Teardown
Run keyword if sys.argv[0].startswith('bin/robot')
...           Remote ZODB TearDown  ${FIXTURE}

Suite Teardown
Run keyword if not sys.argv[0].startswith('bin/robot')
...           Teardown Plone Site
Run keyword if sys.argv[0].startswith('bin/robot')
...           Close all browsers

```

Files of various types can be uploaded to Plone web sites.

Choose file in the *Add new...* menu for a folder to upload a file:

```

*** Test Cases ***

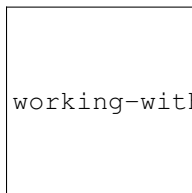
Show add files menu
Go to  ${PLONE_URL}

Wait until element is visible
...  css=span.icon-plone-contentmenu-factories
Click element  css=span.icon-plone-contentmenu-factories
Wait until element is visible
...  css=#plone-contentmenu-factories li.plone-toolbar-submenu-header

Mouse over  file
Update element style  portal-footer  display  none

Capture and crop page screenshot
...  ${CURDIR}/../../_robot/adding-files_add-menu.png
...  css=div.plone-toolbar-container
...  css=#plone-contentmenu-factories ul

```



working-with-content/adding-content/../../_robot/adding-f

Select **File** from the drop-down menu, and you'll see the *Add File* panel:

```
*** Test Cases ***

Show new file add form
    Page should contain element    file
    Click link    file

    Wait until element is visible
    ...    css=#form-widgets-title

    Capture and crop page screenshot
    ...    ${CURDIR}/../../_robot/adding-files_add-form.png
    ...    css=#content
```

working-with-content/adding-content/../../_robot/adding-f

Click the *Browse* button to navigate to the file you want to upload from your local computer. Provide a title (you can use the same file name used on your local computer if you want). Provide a *description* if you want. When you click the save button the file will be uploaded to the folder.

Example file types include PDF files, Word documents, database files, zip files... – well, practically anything. Files on a Plone web site are treated as just files and will show up in contents lists for folders, but there won't be any special display of them. They will appear by name in lists and will be available for download if clicked.

There are specialized add-on tools for Plone web sites that search the content of files, or can provide a preview of for instance PDF or Office files. If you are interested in this functionality, ask your Plone web site administrator.

Adding Links

```
*** Settings ***

Resource    plone/app/robotframework/server.robot
Resource    plone/app/robotframework/keywords.robot
Resource    Selenium2Screenshots/keywords.robot

Library    OperatingSystem

Suite Setup    Run keywords    Suite Setup    Test Setup
Suite Teardown    Run keywords    Test teardown    Suite Teardown

*** Variables ***

${FIXTURE}    plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}    1024    768
@{APPLY_PROFILES}    plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if    not sys.argv[0].startswith('bin/robot')
```

```

...          Setup Plone site   ${FIXTURE}
Run keyword if sys.argv[0].startswith('bin/robot')
...          Open test browser
Run keyword and ignore error Set window size @{DIMENSIONS}

Test Setup
  Import library Remote   ${PLONE_URL}/RobotRemote

  Run keyword if sys.argv[0].startswith('bin/robot')
  ...          Remote ZODB SetUp   ${FIXTURE}

  ${language} = Get environment variable LANGUAGE 'en'
  Set default language   ${language}

  Enable autologin as Manager
  ${user_id} = Translate user_id
  ... default=jane-doe
  ${user_fullname} = Translate user_fullname
  ... default=Jane Doe
  Create user   ${user_id} Member fullname=${user_fullname}
  Set autologin username   ${user_id}

Test Teardown
  Run keyword if sys.argv[0].startswith('bin/robot')
  ...          Remote ZODB TearDown   ${FIXTURE}

Suite Teardown
  Run keyword if not sys.argv[0].startswith('bin/robot')
  ...          Teardown Plone Site
  Run keyword if sys.argv[0].startswith('bin/robot')
  ...          Close all browsers

```

In addition to links embedding within pages, Links can be created as discrete content items. Having links as discrete items lets you do things like organizing them in folders, setting keywords on them to facilitate grouping in lists and search results, or include them in navigation.

Add a link by clicking the menu choice in the *Add new...* menu:

```

*** Test Cases ***

Show add new link menu
  Go to   ${PLONE_URL}


  Wait until element is visible
  ... css=span.icon-plone-contentmenu-factories
  Click element   css=span.icon-plone-contentmenu-factories

  Wait until element is visible
  ... css=#plone-contentmenu-factories li.plone-toolbar-submenu-header

  Mouse over link
  Update element style portal-footer display none

  Capture and crop page screenshot
  ... ${CURDIR}/../../_robot/adding-links_add-menu.png
  ... css=div.plone-toolbar-container
  ... css=#plone-contentmenu-factories ul

```



working-with-content/adding-content/../../../../_robot/adding-l


You will see the Add*Link* panel:

```
*** Test Cases ***

Show new link add form
    Page should contain element  link
    Click link  link

    Wait until element is visible
    ...  css=#form-widgets-IDublinCore-title

    Capture and crop page screenshot
    ...  ${CURDIR}/../../../../_robot/adding-links_add-form.png
    ...  css=#content
```



working-with-content/adding-content/../../../../_robot/adding-l

Good titles for links are important, because the titles will show up in lists of links, and because there tend to be large numbers links held in a folder or collection.

Paste the web address in the URL field or type it in. There is no preview feature here, so it is best to paste the web address from a browser window where you are viewing the target for the link to be sure you have the address correct.

The Link Object in Use

A link object will behave in the following ways, depending on your login status, or permissions.

- **If you have the ability to edit the link object**, when you click on the link object you'll be taken to the object itself so that you can edit it (otherwise you'd be taken to the link's target and could never get to the edit tab!)
- **If you don't have the ability to edit the link object**, when you click on the link object you'll be taken to the target of the link object. Likewise, if you enter the web address of the link object directly in your browser, you'll be taken directly to the link's target. The link object in this case acts as a *redirect*.

Adding Events

```
*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot
```

```

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Open test browser
    Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
    Import library  Remote  ${PLONE_URL}/RobotRemote

    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Remote ZODB SetUp  ${FIXTURE}

    ${language} =  Get environment variable  LANGUAGE  'en'
    Set default language  ${language}

    Enable autologin as  Manager
    ${user_id} =  Translate  user_id
    ...  default=jane-doe
    ${user_fullname} =  Translate  user_fullname
    ...  default=Jane Doe
    Create user  ${user_id}  Member  fullname=${user_fullname}
    Set autologin username  ${user_id}

Test Teardown
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Remote ZODB TearDown  ${FIXTURE}

Suite Teardown
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Teardown Plone Site
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Close all browsers

```

Plone web sites have a built-in system for managing and showing calendar events.

Use the *Add new...* menu for a folder to add an event:

```

*** Test Cases ***

Show add new event menu
    Go to  ${PLONE_URL}

    Wait until element is visible


```

```
... css=span.icon-plone-contentmenu-factories
Click element css=span.icon-plone-contentmenu-factories

Wait until element is visible
... css=#plone-contentmenu-factories li.plone-toolbar-submenu-header

Mouse over event
Update element style portal-footer display none

Capture and crop page screenshot
... ${CURDIR}/../../_robot/adding-events_add-menu.png
... css=div.plone-toolbar-container
... css=#plone-contentmenu-factories ul
```




working-with-content/adding-content/../../_robot/adding-e

Select **Event** from the drop-down menu, and you'll see the rather large *Add Event* panel:

```
*** Test Cases ***

Show new event add form
    Page should contain element event
    Click link event

    Wait until element is visible
    ... css=#mceu_16-body
    Capture and crop page screenshot
    ... ${CURDIR}/../../_robot/adding-events_add-form.png
    ... css=#content
```



working-with-content/adding-content/../../_robot/adding-e

From the top, we have the following fields:

- *Title* - **REQUIRED**
- *Summary*
- *Event starts* - **REQUIRED**
- *Event ends* - **REQUIRED**
- *Whole Day*
- *Open End*
- *Recurrence*
- *Event Location*

- *Attendees*
- *Contact Name*
- *Contact Email*
- *Contact Phone*
- *Event URL*
- *Event body text* (visual editor panel)
- *Change note*

Note: Only three fields, title and start and end date and time, are required.

Although this is a large input panel, if you are in a hurry, type in the title and the start and end times and save. Of course, if you have the other information, you should type it in.

One part of the panel needs a bit more explanation: the event start and end times. Both these can be set using a handy pop-up calendar. This will show when you click on the date.

Setting an event to be “Whole day” will remove the start and end times.

But there are many more options: you can set an event to be “Open-ended” if you don’t know when the end date is, or if it is an ongoing activity that you would still like to show as an event.

For repeating events, use the “recurrence” link. You can set when, and how often, your event will repeat: daily, weekly, every third Tuesday of the month until 2017, etcetera. You can specify that an event should repeat a certain number of times, or until a certain date.

Note: IMPORTANT: Your event will not show on the main web site calendar until it has been **published**.

Adding News Items

```
*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
```

```
...          Setup Plone site  ${FIXTURE}
Run keyword if sys.argv[0].startswith('bin/robot')
...          Open test browser
Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
  Import library  Remote  ${PLONE_URL}/RobotRemote

  Run keyword if sys.argv[0].startswith('bin/robot')
  ...          Remote ZODB SetUp  ${FIXTURE}

  ${language} = Get environment variable  LANGUAGE  'en'
  Set default language  ${language}

  Enable autologin as  Manager
  ${user_id} = Translate  user_id
  ...  default=jane-doe
  ${user_fullname} = Translate  user_fullname
  ...  default=Jane Doe
  Create user  ${user_id}  Member  fullname=${user_fullname}
  Set autologin username  ${user_id}

Test Teardown
  Run keyword if sys.argv[0].startswith('bin/robot')
  ...          Remote ZODB TearDown  ${FIXTURE}

Suite Teardown
  Run keyword if not sys.argv[0].startswith('bin/robot')
  ...          Teardown Plone Site
  Run keyword if sys.argv[0].startswith('bin/robot')
  ...          Close all browsers
```

Plone web sites have a built-in system for publishing news items.

Use the *Add new...* menu for a folder to add a news item:

```
*** Test Cases ***

Show add new news-item menu
  Go to  ${PLONE_URL}

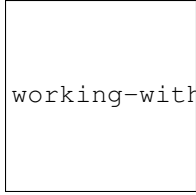
  Wait until element is visible
  ...  css=span.icon-plone-contentmenu-factories
  Click element  css=span.icon-plone-contentmenu-factories

  Wait until element is visible
  ...  css=#plone-contentmenu-factories li.plone-toolbar-submenu-header

  Mouse over  news-item
  Update element style  portal-footer  display  none

  Capture and crop page screenshot
  ...  ${CURDIR}/../../_robot/adding-news-items_add-menu.png
  ...  css=div.plone-toolbar-container
  ...  css=#plone-contentmenu-factories ul
```

You will see the *Add News Item* panel:


 working-with-content/adding-content/../../../../_robot/adding-n

```
*** Test Cases ***
```

```
Show new news-item edit form
```

```
    Page should contain element  news-item
```

```
    Click link  news-item
```

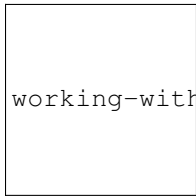
```
    Wait until element is visible
```

```
    ...  css=#mceu_16-body
```

```
    Capture and crop page screenshot
```

```
    ...  ${CURDIR}/../../../../_robot/adding-news-items_add-form.png
```

```
    ...  css=#content
```


 working-with-content/adding-content/../../../../_robot/adding-n

The standard fields for title, description, and change note are in the panel, along with a visual editor area for body text and image and image caption fields. You can be as creative as you want in the body text area, and you can use the insert image (upload image) function to add as much illustration as needed. The images you upload for the news item will be added to the folder in which you are adding the news item.

The *Lead Image* and *Lead Image Caption* fields are for adding an image to be used as a representative graphic for the news item, for posting in news item listings. The image will be automatically resized and positioned. Use the **Body Text** to insert an image in the actual body of the News Item.

Note: IMPORTANT: News items will not appear in the main web site news listing or news portlet until they are published.

Setting Basic Properties

```
*** Settings ***
```

```
Resource  plone/app/robotframework/server.robot
```

```
Resource  plone/app/robotframework/keywords.robot
```

```
Resource  Selenium2Screenshots/keywords.robot
```

```
Library  OperatingSystem
```

```
Suite Setup  Run keywords  Suite Setup  Test Setup
```

```
Suite Teardown  Run keywords  Test teardown  Suite Teardown
```

```
*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Open test browser
    Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
    Import library  Remote  ${PLONE_URL}/RobotRemote

    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Remote ZODB SetUp  ${FIXTURE}

    ${language} =  Get environment variable  LANGUAGE  'en'
    Set default language  ${language}

    Enable autologin as  Manager
    ${user_id} =  Translate  user_id
    ...  default=jane-doe
    ${user_fullname} =  Translate  user_fullname
    ...  default=Jane Doe
    Create user  ${user_id}  Member  fullname=${user_fullname}
    Set autologin username  ${user_id}

Test Teardown
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Remote ZODB TearDown  ${FIXTURE}

Suite Teardown
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Teardown Plone Site
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Close all browsers
```

The tab panels available on each content item has fields for basic information. The more data you can provide, the better Plone can help in making the content available to the relevant visitors.

Any content item, when clicked by a user with edit rights for the item, will show a set of tabs at the top for setting basic properties:

```
*** Test Cases ***

Show basic properties tab
    Go to  ${PLONE_URL}

    Wait until element is visible
    ...  css=span.icon-plone-contentmenu-factories
    Click element  css=span.icon-plone-contentmenu-factories
```

```

Wait until element is visible
... css=#plone-contentmenu-factories li.plone-toolbar-submenu-header

Page should contain element document
Click link document
Update element style portal-footer display none

Wait until element is visible
... css=#form-widgets-IDublinCore-title

Capture and crop page screenshot
... ${CURDIR}/../../_robot/basicproptiestabs.png
... css=nav.autotoc-nav

```

working-with-content/adding-content/../../_robot/basicpro

These basic properties tabs are:

- *Default* - shows the main data entry panel for the content item
- *Settings* - shows a small panel for setting whether or not the item will appear in navigation menus and if comments are allowed on the item
- *Categorization* - shows a panel for creating and setting tags (keywords) for the item
- *Dates* - shows the publishing date and expiration date for the item
- *Ownership* - shows a panel for setting creators, contributors, and any copyright information for the item

The input fields under these tabs cover basic descriptive information called **metadata**. Metadata is sometimes called “data about data.” Plone can use this metadata in a multitude of ways.

Here is the *Categorization* panel, shown for a page content item (would be the same for other content types):

```

*** Test Cases ***

Show edit page categorization
Click link Categorization

Capture and crop page screenshot
... ${CURDIR}/../../_robot/editpagecategorization.png
... css=#content-core

```

working-with-content/adding-content/../../_robot/editpage

The main input field for the panel is for specifying *Tags*, sometimes also known as *Categories*

Create them, just by typing in words or phrases. Plone will automatically search for existing tags while you type, so you can select already existing tags. If you want to create a new tag, just hit “enter” after your word (or phrase).

The *Related Items* field lets you set links between content items, which will show as links at the bottom, when a content item is viewed. This is useful when you don't want to use explicit tags to connect content.

The *Location* field is a geographic location, suitable for use with mapping systems, but appropriate to enter, for general record keeping. Note that this field may not always be present, this is a setting that your site administrator has to enable.

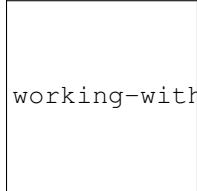
The *Language* choice normally would default to the site setting, but on multilingual web sites, different languages could be used in a mix of content.

The *Dates* panel has fields for the publishing date and the expiration date, effectively start and stop dates for the content if you wish to set them:

```
*** Test Cases ***

Show datessettings
    Click link Dates

    Capture and crop page screenshot
    ... ${CURDIR}/../../../../_robot/datessettings.png
    ... css=#content-core
```



working-with-content/adding-content/../../../../_robot/datesset

The publication and expiration dates work like this:

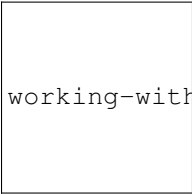
- When an item is past its expiration date, it's marked "expired" in red in its document byline when viewed.
- An item whose publication date is before the current date doesn't get extra text in its byline.
- In both cases, the item is "unpublished", which is not to be confused with a workflow state.
- It merely means the item doesn't show up in listings and searches.
- These listings include folder listings.
- However, the owner of the item will keep seeing it, which is handy because you like to know what you have lying around in your site.
- The permission that controls this is Access inactive portal content.
- Expired items in a folder are marked as such when viewing the folder_contents.
- There's no quick way of seeing if items in a folder listing are not yet published.
- When you set an unpublished item as the default view for a folder, that item will be shown.
- Unpublishing an item doesn't have any effect for admins. They will always see unpublished items in their listings and searches.
- Giving another regular users rights ("can add", "can edit", "can review") on the item doesn't make it any less unpublished for those users.
- A practical way for a non-admin user to access an unpublished item is directly through its URL.

The *Ownership* panel has three free-form fields for listing creators, contributors, and information about copyright or ownership rights to the content:

```
*** Test Cases ***

Show ownershippanel
    Click link Ownership

    Capture and crop page screenshot
    ... ${CURDIR}/../../../../_robot/ownershippanel.png
    ... css=#content-core
```




working-with-content/adding-content/../../../../_robot/ownership

The *Settings* panel has fields that may vary a bit from content type to content type, but generally there are input fields controlling whether or not the item appears in navigation, or if there are comments allowed, and other similar controls:

```
*** Test Cases ***

Show settingspanel
    Click link Settings

    Capture and crop page screenshot
    ... ${CURDIR}/../../../../_robot/settingspanel.png
    ... css=#content-core
```



working-with-content/adding-content/../../../../_robot/settings

You can allow users to edit the “Short name” of content items.

Note: The “Short Name” is part of the URL of a content item. That means that no special characters or spaces are allowed in it. For experienced web editors, it can be handy to manipulate the Short Name directly in order to generate more memorable or shorter URL's.

Recommendations

There is no requirement to enter the information specified through these panels, but it is a good idea to do so. For the *Ownership* panel, providing the data is important for situations where there are several people involved in content creation, especially if there are multiple creators and contributors working in groups. You don't always need fields such as publishing and expiration dates, language, and copyrights, but these data should be specified when appropriate. A content management system can only be as good as the data completeness allows.

Specifying tags requires attention, but if you are able to get in the habit, and are committed to creating a meaningful set of tags, there is a big return on the investment. The return happens through the use of searching and other facilities in Plone that work off the categorization. The same holds for setting related items. You'll be able to put your finger on what you need, and you may be able to discover and use relationships within the content.

Exposing Metadata Properties as meta tags in the HTML source

From Plone 4 on, in *Site Setup*, there is a check box that will expose the *Dublin Core* metadata properties. Checking this box will expose the title, description, etc. metadata as meta tags within the HTML <head>. For example:

```
<meta content="short description" name="DC.description" />
<meta content="short description" name="description" />
<meta content="text/html" name="DC.format" />
<meta content="Page" name="DC.type" />
<meta content="admin" name="DC.creator" />
<meta content="2009-11-27 17:04:03" name="DC.date.modified" />
<meta content="2009-11-27 17:04:02" name="DC.date.created" />
<meta content="en" name="DC.language" />
```

The generator will check and obey the so-called *allowAnonymousViewAbout* setting in the *Control Panel* and affects the properties **Creator**, *Contributors* and *Publisher*.

You can read more about *Dublin Core* and *HTML Metatags*.

Restricting Types in a Folder

```
*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Open test browser
    Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
    Import library  Remote  ${PLONE_URL}/RobotRemote

    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Remote ZODB SetUp  ${FIXTURE}

    ${language} =  Get environment variable  LANGUAGE  'en'
    Set default language  ${language}
```



```

Enable autologin as Manager
${user_id} = Translate user_id
... default=jane-doe
${user_fullname} = Translate user_fullname
... default=Jane Doe
Create user ${user_id} Member fullname=${user_fullname}
Set autologin username ${user_id}

Test Teardown
Run keyword if sys.argv[0].startswith('bin/robot')
... Remote ZODB TearDown ${FIXTURE}

Suite Teardown
Run keyword if not sys.argv[0].startswith('bin/robot')
... Teardown Plone Site
Run keyword if sys.argv[0].startswith('bin/robot')
... Close all browsers

```

The Add new... menu has a choice for restricting the content types that can be added to the folder.

Restricting types available for adding to a folder is the simplest way to control content creation on a Plone web site. You may want to restrict content types if your site is going to be worked on by several people. In this way you can enforce good practices such as putting images in the images folder, or having your “News” items all in the same folder.

Note: Setting restrictions in the very top level, also known as the *root* of the website, is restricted to site administrators. That is because this will influence the navigation, and may lead to unwanted side effects.

First, select the last choice in the *Add new...* menu called *Restrictions...*:

```

*** Test Cases ***
Show restrictions
Go to ${PLONE_URL}/news

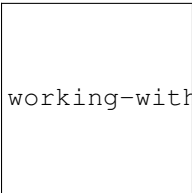
Click link css=#plone-contentmenu-factories a

Wait until element is visible
... css=#plone-contentmenu-factories li.plone-toolbar-submenu-header

Mouse over plone-contentmenu-settings
Update element style portal-footer display none

Capture and crop page screenshot
... ${CURDIR}/../../_robot/show-restrictions.png
... css=div.plone-toolbar-container
... css=#plone-contentmenu-factories ul

```



working-with-content/adding-content/../../_robot/show-res

There are three choices shown for restricting types in the folder:

- Use parent folder settings
- Use portal default
- Select manually

The default choice, to use the setting of the parent folder. That means when you create a folder and restrict the types that can be added, any subfolders created in the folder will automatically carry these restrictions.

The second choice is a way to reset to the default, unrestricted setting.

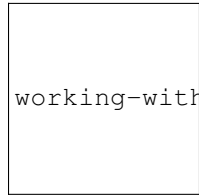
The last choice allows selection from a list of available types:

```
*** Test Cases ***

Menu restrictions
    Go to  ${PLONE_URL}/news/folder_constrainttypes_form

    Click element  form-widgets-constrain_types_mode

    Capture and crop page screenshot
    ...  ${CURDIR}/../../../../_robot/menu-restrictions.png
    ...  css=#main-container
```



working-with-content/adding-content/../../../../_robot/menu-res

Types listed under the *Allowed types* heading are those available on the web site. The default, as shown, is to allow all types. Allowed types may be toggled on and off for the folder.

Use of *Secondary types* allows a kind of more detailed control. For example, if it is preferred to store images in one folder, instead of scattering them in different folders on the web site – a scheme that some people prefer – an “Images” folder could be created with the allowed type set to the Image content type *only*.

Likewise an “Company Events” folder could be created to hold only the Event content type.

If left this way, content creators would be forced (or a single web site owner) to follow this strict scheme.

Perhaps some flexibility is desired for images, though. By checking the Image content type under the *Secondary types* heading for the “Company Events” folder, images could be added if really needed, by using the *More...* submenu, which would appear when this mechanism is in place.

The *Secondary types* will be allowed, but be a little more hidden when adding content. That way, you still have flexibility without confusing part-time editors with too many options.

Some people prefer a heterogeneous mix of content across the web site, with no restrictions. Others prefer a more regimented approach, restricting types in one organizational scheme or another. Plone has the flexibility to accommodate a range of designs.

Preparing Images for the Web

Preparing images for the web is an essential part of using images in Plone, or in any online context. As you will see, size matters.

Many people source photographs taken with a digital camera, but they can also be scanned images, graphical illustrations made with software, and other specialized images. Let’s take a look at the case of a butterfly photo taken with a digital camera.

Digital photographs taken with modern cameras are usually too big to post directly on a website, so they need to be resized. A typical web site design may have a width of around 1000 pixels. When a photograph comes off your camera, it may be several thousand pixels wide and tall, and several megabytes in file size. You need to use software on your computer to resize the image to something less than 1000 x 1000 pixels, often much smaller than that.

The software you use to view or print your digital photos will often have this resizing functionality, or you may have graphics software such as Adobe Photoshop or Gimp on your computer. Resizing an image, sometimes called resampling, is a standard function you should be able to find in your software, often under the *Image* menu. Some also have a special “save for web” command.

How do you know what width, in pixels, to resize your image? It depends. For a little “head shot” photograph to go in a biography, maybe 200 pixels wide is just right. For a group photograph, 200 pixels would be too small to allow identification of the people in the photograph, so it may need to be closer to 400 pixels wide. For a scanned map image, perhaps the image width would need to be 1000 pixels for the map detail to be usable.

After saving your resized image, give it a name that indicates the new size (e.g., butterfly-resized-300px.jpg). The file format is most commonly .jpg (or .jpeg). Other common formats for images include .png and .gif. Take note of where you save images on your computer so that you can find them when you upload them to your Plone web site.

To summarize:

1. Take your photograph with your camera, or find an existing image you want to use
2. Transfer it to your computer
3. Use image software on your computer to resize your photograph
4. Upload it to your Plone website

Adding collections

Collections (formerly called Smart Folders) are virtual containers of lists of items found by doing a specialized search.

See the later section of the manual *Using Listings & Queries (Collections)*

```
*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

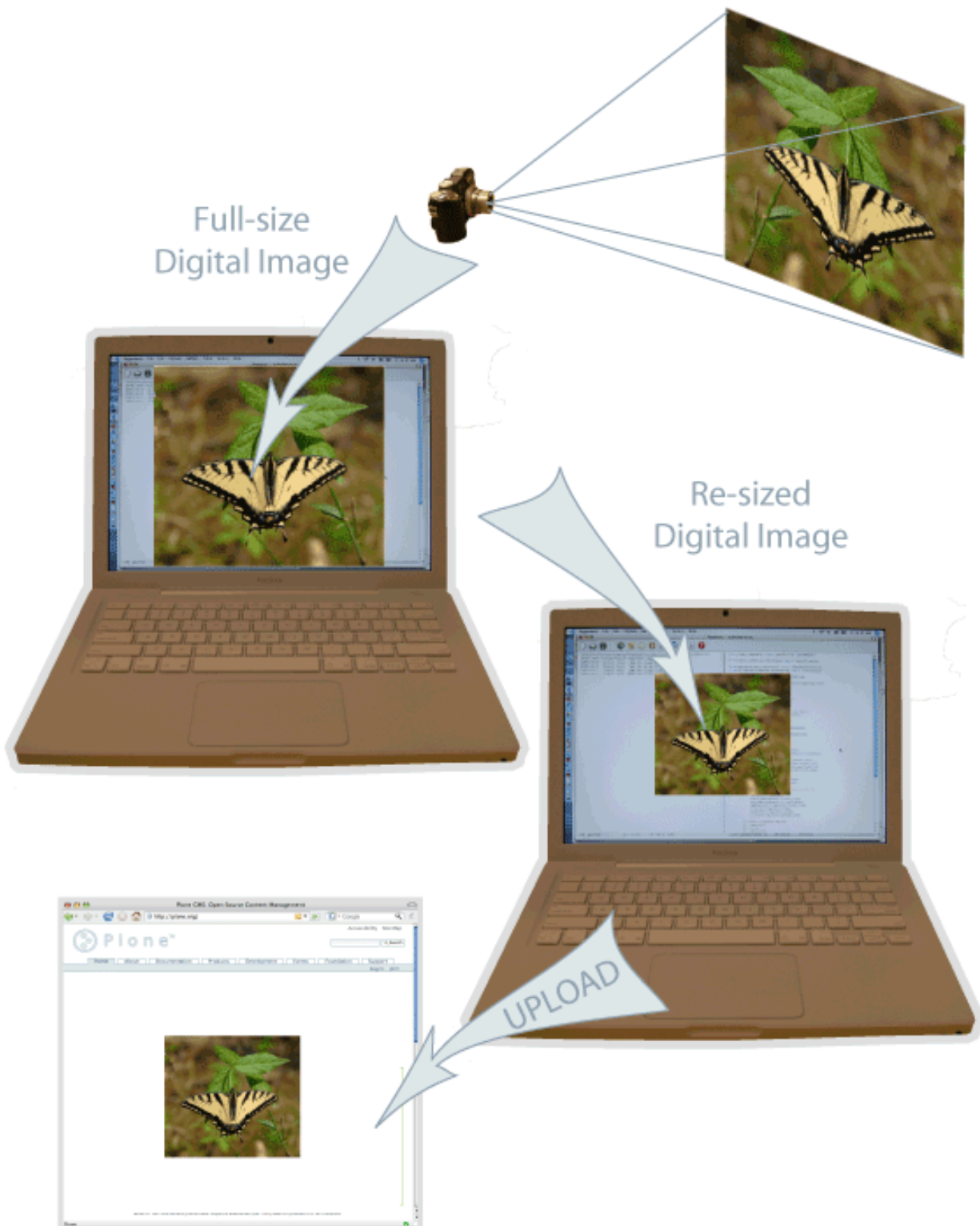
Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
```



```

Run keyword if not sys.argv[0].startswith('bin/robot')
...           Setup Plone site  ${FIXTURE}
Run keyword if sys.argv[0].startswith('bin/robot')
...           Open test browser
Run keyword and ignore error Set window size  @${DIMENSIONS}

Test Setup
  Import library Remote  ${PLONE_URL}/RobotRemote

  Run keyword if sys.argv[0].startswith('bin/robot')
  ...           Remote ZODB SetUp  ${FIXTURE}

  ${language} = Get environment variable LANGUAGE 'en'
  Set default language  ${language}

  Enable autologin as Manager
  ${user_id} = Translate user_id
  ... default=jane-doe
  ${user_fullname} = Translate user_fullname
  ... default=Jane Doe
  Create user  ${user_id} Member fullname=${user_fullname}
  Set autologin username  ${user_id}

Test Teardown
  Run keyword if sys.argv[0].startswith('bin/robot')
  ...           Remote ZODB TearDown  ${FIXTURE}

Suite Teardown
  Run keyword if not sys.argv[0].startswith('bin/robot')
  ...           Teardown Plone Site
  Run keyword if sys.argv[0].startswith('bin/robot')
  ...           Close all browsers

```

Choose “Collection” in the *Add new...* menu for a folder to start defining your collection:

```

*** Test Cases ***

Show add collection menu
  Go to  ${PLONE_URL}
  Wait until element is visible
  ...  css=span.icon-plone-contentmenu-factories
  Click element  css=span.icon-plone-contentmenu-factories
  Wait until element is visible
  ...  css=#plone-contentmenu-factories li.plone-toolbar-submenu-header

  Mouse over  collection
  Update element style portal-footer display none

  Capture and crop page screenshot
  ...  ${CURDIR}/../../_robot/adding-collections_add-menu.png
  ...  css=div.plone-toolbar-container
  ...  css=#plone-contentmenu-factories ul

```

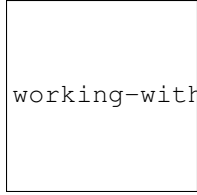
Select **Collection** from the drop-down menu, and you’ll see the *Add Collection* panel:

```

*** Test Cases ***

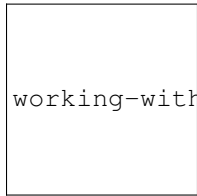
Show new collection add form

```



working-with-content/adding-content/../../_robot/adding-o

```
Page should contain element  collection
Click link  collection
Wait until element is visible
...  css=#mceu_16-body
Capture and crop page screenshot
...  ${CURDIR}/../../_robot/adding-collections_add-form.png
...  css=#content
```

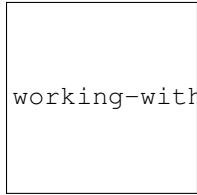


working-with-content/adding-content/../../_robot/adding-o

Apart from the usual fields, the interesting part starts with the **Search terms**

```
*** Test Cases ***

select criteria
  Go to  ${PLONE_URL}/++add++Collection
  Click element  css=div.querystring-criteria-index a
  Capture and crop page screenshot
  ...  ${CURDIR}/../../_robot/collection-criteria.png
  ...  css=div.select2-drop-active
```



working-with-content/adding-content/../../_robot/collecti

You can pick all *meta-data* that Plone has on content items as criteria. By combining more criteria, you can create sophisticated queries, which will be automatically updated.

Your collection can search for all items of types `Page` and `News Item` that have a `Tag` of `Sport`, created in the last 3 months. Or all `Events` that have a `Start date` in the next month.

The possibilities are endless, and Plone will always show the results according to the criteria.

If you create a new content item later with the tag of “Sport”, it will automatically show up in the collection you have just defined.

History

Collections have been around under various names. They used to be called “Smart Folders” in earlier versions of Plone, and you may find references to that in older documentation. It may even be that your site has so-called “Old

Style collections” enabled as well.

See the later section of the manual *Using Listings & Queries (Collections)*

Managing Content

The contents tab is the place where content items can be copied, cut, pasted, moved, renamed, etc.

Editing Content

```
*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Open test browser
    Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
    Import library  Remote  ${PLONE_URL}/RobotRemote

    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Remote ZODB SetUp  ${FIXTURE}

    ${language} =  Get environment variable  LANGUAGE  'en'
    Set default language  ${language}

    Enable autologin as  Manager
    ${user_id} =  Translate  user_id
    ...  default=jane-doe
    ${user_fullname} =  Translate  user_fullname
    ...  default=Jane Doe
    Create user  ${user_id}  Member  fullname=${user_fullname}
    Set autologin username  ${user_id}

Test Teardown
```

```
Run keyword if sys.argv[0].startswith('bin/robot')
...           Remote ZODB TearDown  ${FIXTURE}

Suite Teardown
Run keyword if not sys.argv[0].startswith('bin/robot')
...           Teardown Plone Site
Run keyword if sys.argv[0].startswith('bin/robot')
...           Close all browsers
```

Editing Plone content works the same as adding content – usually the data entry and configuration panels for the content are the same for editing as for adding.

Of course, when we edit an item of content, the item already exists. Click “Edit” on the toolbar when you are viewing it, and you will see the data entry panel for the item, along with the existing values of the item’s data.

For an example of something really simple, where editing looks the same as adding, we can review how to edit the default frontpage on a new Plone site.

The *Edit* panel for a Page shows the title, description and text areas.

Note: If you do wish to give a description, which is a generally a good idea, the description can be text only – there is no opportunity for setting styling of text, such as bold, italics, or other formatting. This keeps the descriptions of Plone content items as simple as possible, and is also required by the *Dublin Core* standard.

```
*** Test Cases ***

Edit folder
Go to  ${PLONE_URL}
Click element  css=#contentview-edit a
Wait until element is visible
...  css=#mceu_16-body
Capture and crop page screenshot
...  ${CURDIR}/../../_robot/edit-page.png
...  css=#content
```



working-with-content/managing-content/../../_robot/edit-p

That’s it. Change what you want, for instance changing the description or the content, and save. The content item will be updated in Plone’s storage system. You can repeatedly edit content items, just as you can repeatedly edit files on your local computer.

Note: Note that there is an extra field, called **Change Note**. Here you can write a short message on why you were editing this, like “updated with our new company motto”. That note will normally not be shown on a public website, but is available for your co-workers to see when they look at the history of a content item.

Cutting, Copying and Pasting Items

```

*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Open test browser
    Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
    Import library  Remote  ${PLONE_URL}/RobotRemote

    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Remote ZODB SetUp  ${FIXTURE}

    ${language} = Get environment variable  LANGUAGE  'en'
    Set default language  ${language}

    Enable autologin as  Manager
    ${user_id} = Translate  user_id
    ...  default=jane-doe
    ${user_fullname} = Translate  user_fullname
    ...  default=Jane Doe
    Create user  ${user_id}  Member  fullname=${user_fullname}
    Set autologin username  ${user_id}

Test Teardown
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Remote ZODB TearDown  ${FIXTURE}

Suite Teardown
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Teardown Plone Site
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Close all browsers

```

Cut, copy, and paste operations involve moving one or more items from one folder to another.

Cut/Paste

Moving items from one area to another on a website is a common task.

It may be that you, or someone else, has created the item in the wrong place. Or, as time goes by, you decide that reordering content will make your site easier to use, for instance in an intranet when projects and their associated files get transferred to another department in the organisation.

Whatever the reason, Plone makes it easy to transfer individual content items, or even whole folders containing hundreds of items, to another location. All internal links will still work. Plone will even redirect external links (where other websites have linked to this content item directly) in most cases. This mechanism can break, however, if you create a new item with the same title and same location as the one you moved.

The easiest way to move content is by using “Contents” on the Toolbar.

```
*** Test Cases ***
Edit folder
  Go to  ${PLONE_URL}
  Click element  css=#contentview-folderContents a
  Capture and crop page screenshot
  ...  ${CURDIR}/../../_robot/foldercontents-cutpaste.png
  ...  css=#content
```



You see an overview of all content in the folder, and in this screenshot that is the content in the top-level or *root* of the site.

You can select individual items, and then use the “Cut” button to cut them. A message “Successfully cut items” will be shown, but the content will still be visible!

Now you can navigate to the folder where you want the content to be, and press the “Paste” button. Only then will the actual moving take place.

The *paste* button remains active, because you would be allowed to continue pasting the content you cut into other places if you wanted. This could happen in several situations, including when you need to copy one page, for example, as a kind of template or basis document, into several folders.

Copy/Paste

A *copy/paste* operation is identical to the *cut/paste* operation, except that there is no removal of content from the original folder. It works as you would expect it to work: the original content remains. If you copy and then paste into the same folder, the (now doubled) new copy of the content will get an automated *short name* of something like “copy_of_originalitem”, which you will most likely want to correct by using the “Rename” button.

Folder View

Folders have the “Display” item on the Toolbar, which controls the different ways of showing folder contents.

```

*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Open test browser
    Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
    Import library  Remote  ${PLONE_URL}/RobotRemote

    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Remote ZODB SetUp  ${FIXTURE}

    ${language} =  Get environment variable  LANGUAGE  'en'
    Set default language  ${language}

    Enable autologin as  Manager
    ${user_id} =  Translate  user_id
    ...  default=jane-doe
    ${user_fullname} =  Translate  user_fullname
    ...  default=Jane Doe
    Create user  ${user_id}  Member  fullname=${user_fullname}
    Set autologin username  ${user_id}

Test Teardown
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Remote ZODB TearDown  ${FIXTURE}

Suite Teardown
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Teardown Plone Site
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Close all browsers

*** Test Cases ***

Show display menu
    Go to  ${PLONE_URL}

```

```
Click link  css=#plone-contentmenu-display a

Wait until element is visible
...  css=#plone-contentmenu-display li.plone-toolbar-submenu-header

Update element style  portal-footer  display  none

Capture and crop page screenshot
...  ${CURDIR}/../../_robot/display-menu.png
...  css=#content-header
...  css=div.plone-toolbar-container
```



working-with-content/managing-content/../../_robot/display-menu.png

For most content items, if you want to change how it looks, you edit the content directly. But folders are a different animal. As containers of other items, folders can display their contents in a variety of ways.

Plone by default comes with these displays:

Standard view A basic list of contained items, showing their title, description, modification date and icon. (The icon and the modification date may or may not be shown to anonymous visitors, depending on your site settings)

Summary view A slightly nicer view, showing the Title and Description and a ‘Read more’ link for each content item. If the contained items have a ‘Lead Image’ declared, like the standard News Item, it will show a thumbnail of that image as well.

Tabular view Compact view, as a table, of the contained items.

All content This displays the full content of each contained item. This can quickly get very long, and is best used only on folders with a small number of items in them, and where each item is in itself not too long. An example could be a list of Frequently Asked Questions, or FAQ.

Album view This view is meant for folders with images in them, it shows a thumbnail for every content item.

Event Listing Designed to give a nice overview of a series of Events.

And often the most useful, yet also more difficult to understand:

“Select a content item as default view”

Note: The standard folder views are often used as the starting point for more elaborate views, that will reflect the purpose of the website. At the very least, these views are often customized with CSS or *Diazo* rules, to show specific information about the content types shown. Many add-ons will provide extra folder views, to properly display any special content items that these add-ons provide. Some widely used add-ons provide a so-called ‘faceted search’ display, which is designed to quickly find your way around large numbers of content.

Therefore, you will likely see more possibilities in this menu if you are working on a real-life site.

Setting an Individual Content Item as the View for a Folder

The basic list view functionality described above for folders fits the normal way we think of folders – as containers of items – but Plone adds a nice facility to set the view of a folder to be that of any single item contained within the folder.

You can set the display view for a folder to show a single page, which can be useful for showing an ‘introductory page’, which you or others have created to explain the purpose of this section of the site, and which contains links to sub-sections and other documents.

Or, you can set it to a “*collection*”, which on its own is already a powerful content filter. One common use-case, and one that is used in a default Plone site, is to have a Folder called “News”. In it, there are individual News items, but also a Collection which will sort all of these news items so the most recent one is shown first.

You would then set the “most recent” collection as the default view of the “News” folder. Using the same ‘Display’ menu, you can then set the display view for the “most recent” to be the “Summary view”

A word of warning

Setting the default view of a folder to be one of the items contained in that folder, is one of the most powerful features of Plone.

But the very fact that you can select for instance a ‘Collection’, which in term displays a whole number of items, as that default item, can also be confusing.

This will play a role when you edit a Folder, which has a content item set as its default view. Or when you want to manage the portlets for a Folder which has a content item set as its default view. Or when you are trying to give *sharing permissions* on a folder, but by accident are only setting them on the default item in that folder.

Plone will display a message in these cases, along the lines of “You are editing the default view of a container. If you wanted to edit the container itself, go *here*.”

Learn to not ignore, but read those messages!

Remember: when setting properties, ask yourself the question: *do I want to do this on the Folder, or just on this one content item that is contained in the Folder?*

Lastly, the display view setting should be used with care, because it can do two things:

- It changes the behavior of folders, from acting as simple containers to acting as direct links to content, if you select an individual content item as default view.
- Changing the view can radically alter the way the information is presented to users. If you select “Album view” on a folder with no images, it may appear the content is gone, since there are no images to show, although of course the information is still there. Or your perfectly written introductory page is suddenly replaced by a boring tabular display.

While the “Display” menu is one of the most powerful features of Plone, it is wise to note for yourself which was the previous view - before you change it - so you can quickly change it back when needed.

Folder Contents

```
*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot
```

```
Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...            Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...            Open test browser
    Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
    Import library  Remote  ${PLONE_URL}/RobotRemote

    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...            Remote ZODB SetUp  ${FIXTURE}

    ${language} =  Get environment variable  LANGUAGE  'en'
    Set default language  ${language}

    Enable autologin as  Manager
    ${user_id} =  Translate  user_id
    ...  default=jane-doe
    ${user_fullname} =  Translate  user_fullname
    ...  default=Jane Doe
    Create user  ${user_id}  Member  fullname=${user_fullname}
    Set autologin username  ${user_id}

Test Teardown
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...            Remote ZODB TearDown  ${FIXTURE}

Suite Teardown
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...            Teardown Plone Site
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...            Close all browsers
```

The Contents item on the Toolbar shows a list of items in a folder. It is the place for simple item-by-item actions and for bulk actions such as copy, cut, paste, move, reorder, etc.

The Contents tab for folders is like “File Manager” or “My Computer” system utilities in Windows and Linux desktops and the “Finder” in Mac OS X, with similar functionality.

```
*** Test Cases ***

Edit folder
    Go to  ${PLONE_URL}
```

```

Click element  css=#contentview-folderContents a
Capture and crop page screenshot
...  ${CURDIR}/../../../../_robot/foldercontents.png
...  css=#content
Capture and crop page screenshot
...  ${CURDIR}/../../../../_robot/foldercontents-columns.png
...  css=#btn-attribute-columns
Capture and crop page screenshot
...  ${CURDIR}/../../../../_robot/foldercontents-selected.png
...  css=#btn-selected-items
Capture and crop page screenshot
...  ${CURDIR}/../../../../_robot/foldercontents-rearrange.png
...  css=#btn-rearrange
Capture and crop page screenshot
...  ${CURDIR}/../../../../_robot/foldercontents-rearrange.png
...  css=#btn-rearrange
Capture and crop page screenshot
...  ${CURDIR}/../../../../_robot/foldercontents-upload.png
...  css=#btn-upload
Capture and crop page screenshot
...  ${CURDIR}/../../../../_robot/foldercontents-cut.png
...  css=#btn-cut
Capture and crop page screenshot
...  ${CURDIR}/../../../../_robot/foldercontents-copy.png
...  css=#btn-copy
Capture and crop page screenshot
...  ${CURDIR}/../../../../_robot/foldercontents-paste.png
...  css=#btn-paste
Capture and crop page screenshot
...  ${CURDIR}/../../../../_robot/foldercontents-delete.png
...  css=#btn-delete
Capture and crop page screenshot
...  ${CURDIR}/../../../../_robot/foldercontents-rename.png
...  css=#btn-rename
Capture and crop page screenshot
...  ${CURDIR}/../../../../_robot/foldercontents-tags.png
...  css=#btn-tags
Capture and crop page screenshot
...  ${CURDIR}/../../../../_robot/foldercontents-state.png
...  css=#btn-workflow
Capture and crop page screenshot
...  ${CURDIR}/../../../../_robot/foldercontents-properties.png
...  css=#btn-properties
Capture and crop page screenshot
...  ${CURDIR}/../../../../_robot/foldercontents-searchbox.png
...  css=#filter

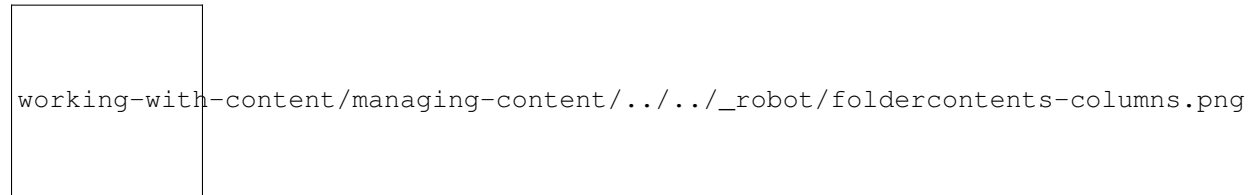
```

working-with-content/managing-content/../../../../_robot/folder

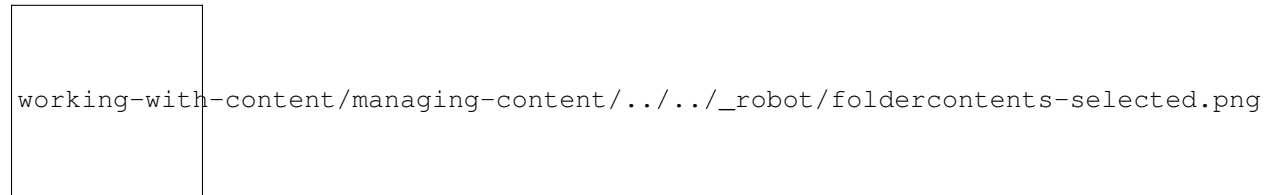
The general method is to select one or more items, by checking the checkbox in front of their name, and then performing the desired operation.

Shift-clicking to select a range of items works. This could be very handy for a folder with more than a dozen items or so, and would be indispensable for folders with hundreds of items.

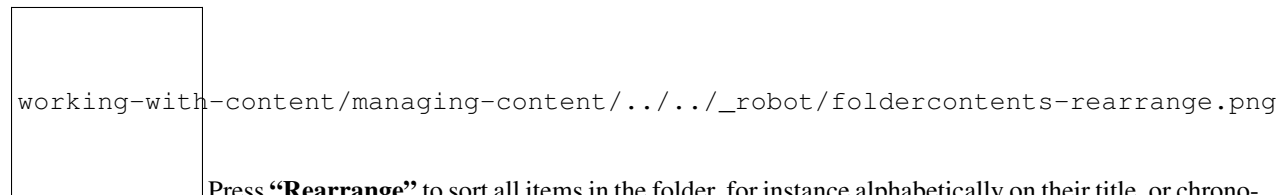
Let’s go over the possibilities one by one:



The first icon lets you select the columns to show. This can help you find the right content.

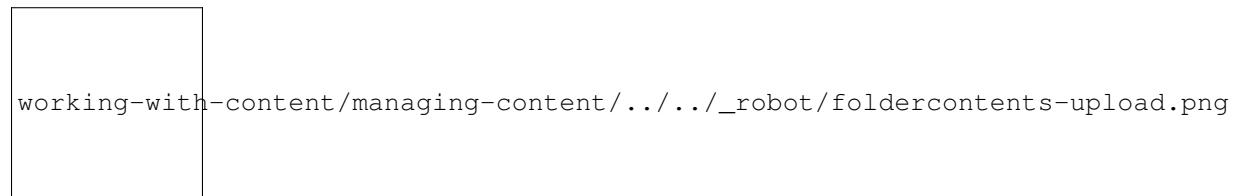


The second item in the horizontal bar shows how many items you have selected.

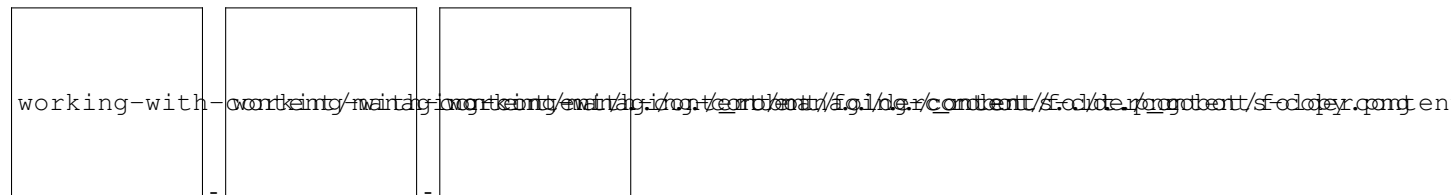


Press **“Rearrange”** to sort all items in the folder, for instance alphabetically on their title, or chronologically by creation date, published date or date last modified.

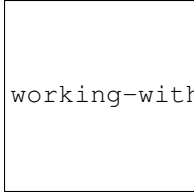
Warning: Be careful when using this option, especially in the root of the site. As folders get re-arranged, it will change the order of the navigation tabs in your site.



“Upload” allows you to upload one or more files (like images or PDF’s) from your computer.

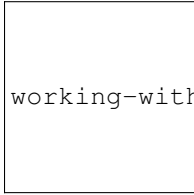


“Cut”, **“Copy”** and **“Paste”** do what you *expect them to do*.



working-with-content/managing-content/../../../../_robot/foldercontents-delete.png

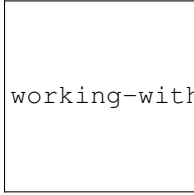
The “Delete” button has a red color, since this is a potentially *dangerous operation*.



working-with-content/managing-content/../../../../_robot/foldercontents-rename.png

Rename will open up a form where you can change the Title and the *short name* for an item. The **Title** can be anything you like, but the **short name** is part of the URL. That means you have to abide by certain rules:

- it cannot contain any spaces or special characters like * or \. When you create an item, Plone generates a safe *short name* from the Title, but when you change this later you should take care this remains a valid URL.
- it has to be unique in a folder. You can have two items with the same Title (although it would be confusing), but you cannot have two items with the same *short name* within the same folder. It’s perfectly fine to have the same *short name* being used in different folders.



working-with-content/managing-content/../../../../_robot/foldercontents-tags.png

Tags allows you to set tags on several items in bulk. This can be a real time-saver.



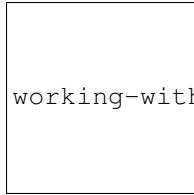
working-with-content/managing-content/../../../../_robot/foldercontents-state.png

State will allow you to change the workflow state of one or more items, such as *publishing* them. See the [chapter on collaboration and workflow](#) for in-depth information.



working-with-content/managing-content/../../../../_robot/foldercontents-properties.png

With the **Properties** button you can set things like the *Publication Date*, *Expiration Date*, copyright and other metadata on your content items.



working-with-content/managing-content/../../../../_robot/foldercontents-searchbox.png

And finally, use the Query searchbox to locate content items, if you know (part of) the title or any other identifier. This is a very quick way to get to content items in folders with hundreds or thousands of items.

Reordering Items

```
*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Open test browser
    Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
    Import library  Remote  ${PLONE_URL}/RobotRemote

    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Remote ZODB Setup  ${FIXTURE}

    ${language} =  Get environment variable  LANGUAGE  'en'
    Set default language  ${language}

    Enable autologin as  Manager
    ${user_id} =  Translate  user_id
    ...  default=jane-doe
    ${user_fullname} =  Translate  user_fullname
    ...  default=Jane Doe
    Create user  ${user_id}  Member  fullname=${user_fullname}
    Set autologin username  ${user_id}

Test Teardown
```

```

Run keyword if sys.argv[0].startswith('bin/robot')
...           Remote ZODB TearDown  ${FIXTURE}

Suite Teardown
Run keyword if not sys.argv[0].startswith('bin/robot')
...           Teardown Plone Site
Run keyword if sys.argv[0].startswith('bin/robot')
...           Close all browsers

```

Using “Contents” on the Toolbar gives you the overview of a folder. From here, you can do manual reordering of items in a folder.

```

*** Test Cases ***

Edit folder
Go to  ${PLONE_URL}
Click element  css=#contentview-folderContents a
Capture and crop page screenshot
...  ${CURDIR}/../../_robot/foldercontents-reorder.png
...  css=#content

```

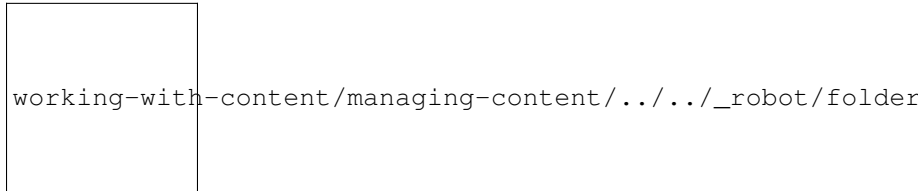


Fig. 3.1: Simply hover over the content item you want to reorder (any column is fine, just don’t hover exactly over the title), and the cursor changes into a hand. Click and drag to reorder.

Next/Previous Navigation

Automatic previous-next links for content items in a folder can be enabled under the Settings tab for a folder.

The *Settings* tab is found by clicking the *Edit* tab for the folder.

Once enabled, as content items are added to the folder, previous and next links will automatically appear at the bottom of each content item.

This is a *really* useful feature!

Deleting Items

```

*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

```

```
*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...            Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...            Open test browser
    Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
    Import library  Remote  ${PLONE_URL}/RobotRemote

    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...            Remote ZODB SetUp  ${FIXTURE}

    ${language} = Get environment variable  LANGUAGE  'en'
    Set default language  ${language}

    Enable autologin as  Manager
    ${user_id} = Translate  user_id
    ...  default=jane-doe
    ${user_fullname} = Translate  user_fullname
    ...  default=Jane Doe
    Create user  ${user_id}  Member  fullname=${user_fullname}
    Set autologin username  ${user_id}

Test Teardown
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...            Remote ZODB TearDown  ${FIXTURE}

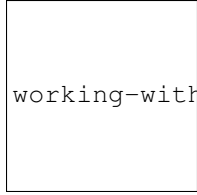
Suite Teardown
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...            Teardown Plone Site
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...            Close all browsers
```

Items may be deleted from a folder with ease.

Sometimes it is necessary to delete a content item. Again, the easiest way to do this is by using “Contents” on the Toolbar.

```
*** Test Cases ***

Edit folder
    Go to  ${PLONE_URL}
    Click element  css=#contentview-folderContents a
    Capture and crop page screenshot
    ...  ${CURDIR}/../../_robot/foldercontents-delete.png
    ...  css=#content
```



working-with-content/managing-content/../../../../_robot/folder

Simply select the content item(s) you want to delete, and press the “Delete” button.

Entire folders may be deleted, so care must be taken with the delete operation!

Plone will warn you if there are other content items in your site that link to the one you are deleting, and will offer to open those in a separate window so you can edit them.

Note: Plone’s database will keep a record of deleted items, and in many cases your site administrator will be able to undo an accidental delete. However, this will only work if there haven’t been many site edits after the delete operation, so if you accidentally delete content, it is important to act soon. Careful site administrators will also have regular backups of the entire site, but getting your content back from there will involve more work. An alternative to deleting content is to ‘un-publish’ it, so it isn’t available to outside visitors anymore.

Automatic Locking and Unlocking

Plone gives you a locking message that will tell you that a document was locked, by whom, and how long ago - so you won’t accidentally stomp on somebody else’s changes.

When somebody clicks on the Edit tab, that item immediately becomes locked. This feature prevents two people from editing the same document at the same time, or accidentally saving edits over another user’s edits.

If a user leaves the edit page without clicking Save or Cancel, the content locking will remain effective for the next ten minutes after which time, the locked content item becomes automatically unlocked.

This timeout feature is important for some browsers that do not execute the “on-unload” javascript action properly, such as Safari. Should you desire to disable locking, go to the Plone control panel (Site Setup -> Site) and uncheck *Enable locking for through-the-web edits*.

Finally, users with the role “Site-Admin” can override the lock and unlock it.

Versioning

An overview on how to view the version history of an item, compare versions, preview previous versions and revert to previous versions.

Creating a new version

Plone includes a versioning feature. By default, the following content types have versioning enabled:

- Pages
- News Items
- Events
- Links

Note that all other content types do track workflow history (so, when an item was published, unpublished etcetera)

Content items can be configured to have versioning enabled/disabled through the Site Setup Plone Configuration panel under “Types”.

When editing an item, you may use the **change note** field at the bottom; the change note will be stored in the item’s version history. If the change note is left blank, Plone includes a default note: “Initial Revision”.

A new version is created every time the item is saved. Versioning keeps track of all kinds of edits: content, metadata, settings, etc.

Viewing the version history

Once an item has been saved, you can see the **History** by clicking on the *clock* item in the Toolbar.

```
*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Open test browser
    Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
    Import library  Remote  ${PLONE_URL}/RobotRemote

    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Remote ZODB Setup  ${FIXTURE}

    ${language} =  Get environment variable  LANGUAGE  'en'
    Set default language  ${language}

    Enable autologin as  Manager
    ${user_id} =  Translate  user_id
    ...  default=jane-doe
    ${user_fullname} =  Translate  user_fullname
    ...  default=Jane Doe
    Create user  ${user_id}  Member  fullname=${user_fullname}
    Set autologin username  ${user_id}
```

```

Test Teardown
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...            Remote ZODB TearDown  ${FIXTURE}

Suite Teardown
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...            Teardown Plone Site
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...            Close all browsers

```

```

*** Test Cases ***

Create sample content
    Go to  ${PLONE_URL}

    ${item} = Create content  type=Document
    ... id=samplepage  title=Sample Page
    ... description=The long wait is now over
    ... text=<p>Our new site is built with Plone.</p>
    Fire transition  ${item}  publish

    Go to  ${PLONE_URL}/samplepage
    Click element  css=#contentview-edit  a
    Click element  css=#form-widgets-IDublinCore-title
    Input text  css=#form-widgets-IDublinCore-title  Hurray
    Click element  css=#form-widgets-IVersionable-changeNote
    Input text  css=#form-widgets-IVersionable-changeNote  Title should be Hurray,
↳not Sample Page.
    Click button  css=#form-buttons-save

Show history
    Go to  ${PLONE_URL}/samplepage
    Click link  css=#contentview-history  a
    Wait until element is visible
    ...  css=#history-list
    Update element style  portal-footer  display  none

    Capture and crop page screenshot
    ...  ${CURDIR}/../../_robot/content-history.png
    ...  css=#content-header
    ...  css=div.plone-toolbar-container

```

working-with-content/managing-content/../../_robot/content

The most recent version is listed first. The History view provides the following information:

- The type of edit (content or workflow)
- Which user made the edit
- What date and time the edit occurred

In the above example, Jane created a Page, then published it. Then, she decided to edit the Page, change it's title and she put in "Title should be Hurray, not Sample Page." in the "Change notes" box. Here you can see why it's good to put in change notes: you get a good overview of *why* an item was edited.

Comparing versions

From the History viewlet you can compare any previous version with the current version or any other version with the version just before it.

To compare any previous version with the one just before it, click the *Compare* link located between two adjacent versions in the History overlay.

By clicking this button, you'll see a screen like this one where you can see the differences between the two versions:

You may also compare any previous version to the *current* version by clicking the *Compare to current* link.

Viewing and reverting to previous versions

You can preview any previous version of a document by clicking the *View* link to the right of any version listed.

To revert back to a previous version, click on the *Revert to this revision* button to the right of any version listed.

Working Copy

```
*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{CONFIGURE_PACKAGES}  plone.app.iterate
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content  plone.app.iterate:plone.app.
↪iterate
${REGISTER_TRANSLATIONS}  ${CURDIR}/../../_locales

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Open test browser
    Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
    Import library  Remote  ${PLONE_URL}/RobotRemote
```



```

Run keyword if sys.argv[0].startswith('bin/robot')
...           Remote ZODB SetUp  ${FIXTURE}

${language} = Get environment variable  LANGUAGE  'en'
Set default language  ${language}

Enable autologin as  Manager
${user_id} = Translate  user_id
... default=jane-doe
${user_fullname} = Translate  user_fullname
... default=Jane Doe
Create user  ${user_id}  Member  fullname=${user_fullname}
Set autologin username  ${user_id}

Test Teardown
Run keyword if sys.argv[0].startswith('bin/robot')
...           Remote ZODB TearDown  ${FIXTURE}

Suite Teardown
Run keyword if not sys.argv[0].startswith('bin/robot')
...           Teardown Plone Site
Run keyword if sys.argv[0].startswith('bin/robot')
...           Close all browsers

```

Working Copy lets you have two versions of your content in parallel.

Note: When a Plone site is first created, there are a number of additional features that can be enabled, including “Working Copy”. If the Plone site you are using doesn’t show the “Check out” option under the Actions menu, you will need to contact your site manager and request that “Working Copy Support (Iterate)” be installed.

Overview

You might have been in a situation like this before: you have published a document, and you need to update it extensively, but you want the old version to exist on the web site until you have published the new one. You also want the new document to replace the current one, but you’d like to keep the history of the old one, just in case.

Working copy makes all this possible.

Essentially, you “check out” a version of the currently published document, which creates a “working copy” of the document. You then edit the working copy (for as long as you like) and when you’re ready for the new version to go live, you “check in” your working copy, and it’s live.

Behind the scenes, Plone will replace the original document with the new one in the exact same location and web address and archive the old version as part of the document’s version history.

Using “Check out”

First, navigate to the page you want to check out. Then from the “Actions” menu, select “Check out”:

```
*** Test Cases ***
```

```
Show how to checkout
  Go to  ${PLONE_URL}/front-page

  Wait until element is visible
  ...  css=span.icon-plone-contentmenu-actions
  Click element  css=span.icon-plone-contentmenu-actions
  Wait until element is visible
  ...  css=#plone-contentmenu-actions li.plone-toolbar-submenu-header

  Mouse over  css=#plone-contentmenu-actions-iterate_checkout
  Update element style  portal-footer  display  none

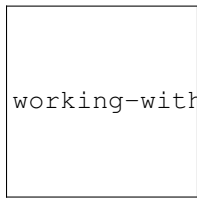
  Capture and crop page screenshot
  ...  ${CURDIR}/../../_robot/working-copy_checkout.png
  ...  css=#content-header
  ...  css=div.plone-toolbar-container

Show checkout notification

  Go to  ${PLONE_URL}/front-page

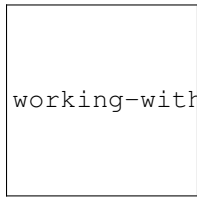
  Wait until element is visible
  ...  css=span.icon-plone-contentmenu-actions
  Click element  css=span.icon-plone-contentmenu-actions
  Wait until element is visible
  ...  css=#plone-contentmenu-actions li.plone-toolbar-submenu-header
  Click link  css=#plone-contentmenu-actions-iterate_checkout
  Wait until element is visible
  ...  name=form.button.Checkout
  Click button  name=form.button.Checkout
  Element should be visible  css=.portalMessage
  Update element style  portal-footer  display  none

  Capture and crop page screenshot
  ...  ${CURDIR}/../../_robot/working-copy_checkout-notification.png
  ...  css=#content-header
  ...  css=div.plone-toolbar-container
```



working-with-content/managing-content/../../_robot/workin

An info message will appear indicating you're now working with a working copy:



working-with-content/managing-content/../../_robot/workin

Now you're free to edit your own local copy of a published document. During this time, the original document is "locked" – that is, no one else can edit that published version while you have a working copy checked out. This will

prevent other changes from being made to (and subsequently lost from) the published version while you edit your copy.

working-with-content/managing-content/../../../../_robot/working

```
Show locked original
Go to  ${PLONE_URL}/front-page

Element should be visible  css=#plone-lock-status
Update element style  portal-footer  display  none

Capture and crop page screenshot
...  ${CURDIR}/../../../../_robot/working-copy_locked.png
...  css=#content-header
...  css=div.plone-toolbar-container
```

Using “Check in”

When you are ready to have your edited copy replace the published one, simply choose “Check-in” from the “Actions” drop-down menu:

working-with-content/managing-content/../../../../_robot/working

```
Show check-in option
Go to  ${PLONE_URL}/copy_of_front-page

Wait until element is visible
...  css=span.icon-plone-contentmenu-actions
Click element  css=span.icon-plone-contentmenu-actions
Wait until element is visible
...  css=#plone-contentmenu-actions li.plone-toolbar-submenu-header

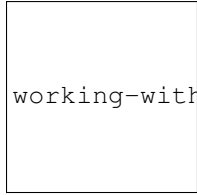
Mouse over  css=#plone-contentmenu-actions-iterate_checkin
Update element style  portal-footer  display  none
Capture and crop page screenshot
...  ${CURDIR}/../../../../_robot/working-copy_checkin.png
...  css=#content-header
...  css=div.plone-toolbar-container

Click link  css=#plone-contentmenu-actions-iterate_checkin

Element should be visible  css=#checkin_message
Update element style  portal-footer  display  none
```

```
Capture and crop page screenshot
...  ${CURDIR}/../../../../_robot/working-copy_checkin-form.png
...  css=#content-header
...  css=div.plone-toolbar-container
```

You will then be prompted to enter a Check-in message. Fill it out and click on “Check in”:



working-with-content/managing-content/../../../../_robot/workin

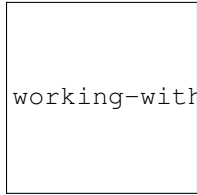
Your updated document will now replace the published copy and become the new published copy.

You will also notice that there is no longer a copy of the document in the folder.

Note that it is not necessary (and in fact, it is not recommended) to use the “State” drop-down menu on a working copy. If you inadvertently do so, however, don’t panic. Just go back to your working copy and use “Check in” from the “Actions” menu.

Canceling a “Check out”

If for any reason it becomes necessary to cancel a check out and **you don’t want to save any of your changes**, simply navigate to the working copy and select “Cancel check-out”:



working-with-content/managing-content/../../../../_robot/workin

```
Show cancel checkout
Go to  ${PLONE_URL}/copy_of_front-page

Wait until element is visible
...  css=span.icon-plone-contentmenu-actions
Click element  css=span.icon-plone-contentmenu-actions
Wait until element is visible
...  css=#plone-contentmenu-actions li.plone-toolbar-submenu-header

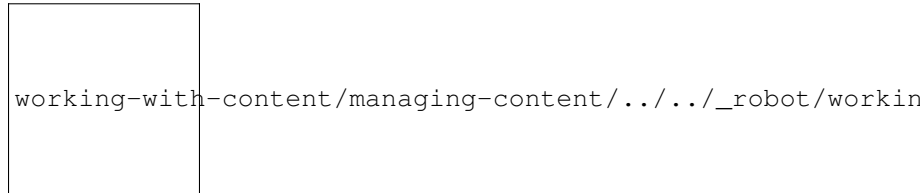
Mouse over  css=#plone-contentmenu-actions-iterate_checkout_cancel
Update element style  portal-footer  display  none
Capture and crop page screenshot
...  ${CURDIR}/../../../../_robot/working-copy_cancel-checkout.png
...  css=#content-header
...  css=div.plone-toolbar-container

Click link  css=#plone-contentmenu-actions-iterate_checkout_cancel

Element should be visible  css=.destructive
Update element style  portal-footer  display  none
```

```
Capture and crop page screenshot
...  ${CURDIR}/../../../../_robot/working-copy_cancel-checkout-form.png
...  css=#content-header
...  css=div.plone-toolbar-container
```

You will prompted to confirm the “Cancel checkout” or to “Keep checkout”:



Note: If the user who has checked out a working copy is not available to check in or cancel a check out, users with the Manager role may navigate to the working copy and perform either the check in or cancel check out actions. That’s because not all contributors have the *Check in* privilege. If that option is missing from your *Actions* menu:

1. Use the *State* menu.
2. Submit for publication.
3. Ask a reviewer to **not** change the state.
4. Ask the reviewer to perform the check in on your behalf instead.

The check in routine will handle the state.

Using Content Rules

This tutorial discusses what content rules are and how to configure and use them.

Helicopter view

Content Rules are a powerful mechanism to automate dealing with content.

There are several steps, that follow one another:

Define the rule

1. create and name a new rule, specifying what triggers it to be executed
2. then narrow it down, by using conditions, to execute only on the content items you want
3. and select the action or actions that should be taken

Afterwards, you end up with a ready-to-go, named Content Rule. But by default it will not execute yet; there is one more step to take:

Apply it to a portion of your site. Or to the whole site, if you so choose to.

Why is the setting up of a Content Rule separated from the application? Because you may want to have different content rules applied to different parts of your site.

Triggers, conditions, actions

A general overview what makes up a content rule, some sample use cases, and who can set up and use content rules.

What is a content rule?

A content rule will automatically perform an action when certain events (known as “triggers”) take place.

You can set up a content rule to send an email (the action) whenever certain (or any) content is added to a specific folder (the trigger).

Other use cases for content rules

- Move content from one folder to another when that content item is published
- Send email when a content item is deleted
- Delete content after a certain date

Who can set up and use content rules?

Site Manager permissions are required to in order to set up and apply content rules.

What are the triggers, conditions and actions that come with Plone 5.0?

The following general **triggers** are available by default:

- Comment added
- Comment removed
- Comment reply added
- Comment reply removed
- Object added to this container
- Object copied
- Object modified
- Object removed from this container
- User created
- User logged in
- User logged out
- User removed
- Workflow state changed

These triggers can be made more specific by using **conditions**

The following general **conditions** are available by default:

- Content type: the type has to be one or more specific ones, like a Page or News Item
- File extension: do only for type .PDF, for instance
- Workflow state: only act on unpublished items, for instance
- Workflow transition: only act when an item is being published
- User’s Group: only act when one of the “Sports Team” members logs in
- User’s Role: only act when a Site Administrator logs in
- TALES expression: an advanced, programmable condition.

The following **actions** are available by default:

- Logger - make an entry in the event log
- Notify user with an information, warning, or error message
- Copy to folder
- Move to folder
- Delete object
- Transition workflow state
- Send email

Note: Content Rules are extendable.

There are add-ons available that will create new actions, conditions or triggers, and you can also write your own.

Creating and Defining Content Rules

How to define content rules using the triggers and actions included in Plone

Creating a Rule

Content rules are created globally through the Plone Control Panel (“Site Setup” link) and then selected from the Rules tab for the desired folder (or at the Plone site root if you want the rule applied site-wide).

In this example, you’re going to create a content rule that will send an email any time a Page type is modified.

- Click on “Content Rules” from the Site Setup page
- The first option, “Disable Globally”, allows you to disable ALL content rules. This is the emergency brake, for when you have created rules that are not doing what you want.
- In the second section of the main page for Content Rules is where any existing content rules will be listed. There are also some checkboxes to filter them, which can be useful if there are many rules defined.

```
*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***
```

```
Suite Setup
    Run keyword if not sys.argv[0].startswith('bin/robot')
    ...           Setup Plone site  ${FIXTURE}
    Run keyword if sys.argv[0].startswith('bin/robot')
    ...           Open test browser
    Run keyword and ignore error Set window size  @${DIMENSIONS}
```

```
Test Setup
    Import library Remote  ${PLONE_URL}/RobotRemote

    Run keyword if sys.argv[0].startswith('bin/robot')
    ...           Remote ZODB SetUp  ${FIXTURE}

    ${language} = Get environment variable LANGUAGE 'en'
    Set default language  ${language}

    Enable autologin as Manager
    ${user_id} = Translate user_id
    ... default=jane-doe
    ${user_fullname} = Translate user_fullname
    ... default=Jane Doe
    Create user  ${user_id} Member fullname=${user_fullname}
    Set autologin username  ${user_id}
```

```
Test Teardown
    Run keyword if sys.argv[0].startswith('bin/robot')
    ...           Remote ZODB TearDown  ${FIXTURE}
```

```
Suite Teardown
    Run keyword if not sys.argv[0].startswith('bin/robot')
    ...           Teardown Plone Site
    Run keyword if sys.argv[0].startswith('bin/robot')
    ...           Close all browsers
```

```
*** Test Cases ***
```

```
Show contentrules
```

```
    Go to  ${PLONE_URL}/@@rules-controlpanel
```

```
    Capture and crop page screenshot
```

```
    ...  ${CURDIR}/../../../../_robot/contentrules-start.png
```

```
        ...  css=#content
```

```
        ...  css=div.plone-toolbar-container
```

```
add rule
```

```
    Go to  ${PLONE_URL}/+rule/plone.ContentRule
```

```
    Wait until element is visible
```

```
    ...  css=#formfield-form-widgets-title
```

```
    Click element  css=#form-widgets-title
```

```
    Input text  css=#form-widgets-title Send Email when any Page is Modified
```

```
    Click element  css=#form-widgets-description
```

```
    Input text  css=#form-widgets-description this rule is meant for folders where_
↪new staff is having a go
```

```
    Click element  css=#formfield-form-widgets-event
```

```
    Select From List  id=form-widgets-event Object modified
```



```

Capture and crop page screenshot
...  ${CURDIR}/../../../../_robot/contentrules-add.png
    ...  css=#content
Click button  css=#form-buttons-save
Capture and crop page screenshot
...  ${CURDIR}/../../../../_robot/contentrules-conditions.png
    ...  css=#content
Wait until element is visible
...  name=form.button.Save
Click button  name=form.button.Save

assign rule
  Go to  ${PLONE_URL}/news
  Click link  css=#contentview-contentrules a
  Update element style  portal-footer  display  none

  Capture and crop page screenshot
  ...  ${CURDIR}/../../../../_robot/contentrules-assign.png
    ...  css=#content
    ...  css=div.plone-toolbar-container

```

working-with-content/managing-content/../../../../_robot/content

If no content rules exist, the only option is an “Add content rule” button. Click that.

An “Add Rule” form comes up. Enter a descriptive title – for this example, use: “Send Email when any Page is Modified”. Enter a description if desired.

For the “Triggering event” select “Object modified”. Leave “Enabled” checked, and “Stop executing rules” and “Cascading rule” unchecked.

working-with-content/managing-content/../../../../_robot/content

Click the “Save” button. At this point, you have essentially created a “container” for the content rule.

Next you’ll further define the trigger and actions for this rule.

working-with-content/managing-content/../../../../_robot/content

Defining conditions and actions

After creating a content rule, you need to actually define the specific conditions of the trigger and actions that will occur based on those conditions.

For the condition:

- By default, “Content type” is selected and since you want a trigger only for Pages, click on the “Add” button.
- From the “Add Content Type Condition” page, select “Page” and click on “Save”

For the action:

- Select “Send email” from the drop down menu and click on the “Add” button.
- From the “Add Mail Action” page, fill out the form:
- For the “Subject” enter: “Automated Notification: Page Modified”
- “Email source” is the From: address and is optional
- “Email recipients” is the To: address; enter a valid email address
- For the “Message” enter what you want for the body of the email
- Click the “Save” button

Congratulations, you have created a working content rule!

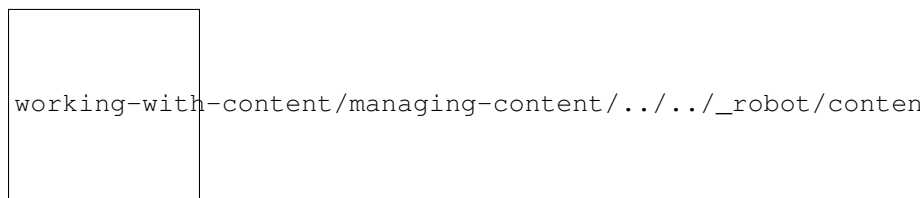
In the next section, you’ll learn how easy it is to apply this content rule to any part (or all) of your Plone site.

Assigning a Content Rule

Now that you’ve set up a content rule, how does it actually get used?

At this point, you have successfully created a content rule. However, this content rule isn’t actually in use until it has been assigned and enabled on one or more folders.

- Navigate to the folder where you want the content rule to be in effect. This can be any folder on the Plone site or it can be for the entire Plone site (“Home”). In this example we’re going to the “News” folder.
- Click on the “Rules” tab. From there you will see a drop down menu of possible content rules:



- Select the desired content rule (“Send Email...” in this example) and click on the “Add” button.
- By default, the rule has now been applied to the current folder only as indicated by the symbol in the “Enabled here” column indicates.

There will be several buttons near the bottom.

Tick the check box for the rule you want (“Send Email...”) and then click on either “Apply to subfolders” button.

Now this content rule will also apply to any subfolder that exist now or are created in the future.

If you wish to have this rule apply to all the subfolders but not to the current folder, then tick the check box next to the rule and click on the “Disable” button.

Note: The “Enabled here” column is empty for this rule now.

You will need to explicitly use the “Enable” button to re-active this rule for the current folder; using the “Apply to current folder only” button will **NOT** re-enable the content rule.

The `Apply to subfolders` and `Apply to current folder only` can be thought of as toggles.

You can test this rule now by creating a new Page or modifying an existing Page.

Once you click on `Save` for that Page, an email will be sent.

Managing Multiple Rules

For each rule, you can define if additional rules should be applied after it, or if it is the end of the pipeline.

Furthermore, you can specify if you want rules to be *cascading* or not.

An example: the first `ContentRule` is triggered for a content item, which is then moved to a folder. But, in this folder, another `ContentRule` is active, which will operate on any new content item that gets moved into that folder. *Cascade* means that yes, the second rule should be applied.

Note: It is entirely possible to create never-ending loops this way: Rule1 moves newly published files in Folder1 to Folder2.

But Rule2, which is active in Folder2, unpublishes any newly moved files, and then copies them into Folder1. And so it goes round and round...

Be careful when using cascading rulesets!

“Navigating” with assigned content rules

The “Edit Content Rule” page uses a ‘related items’ like display (“Assignments”) for listing all the locations where the rule is assigned. From there, you can go directly to that folder’s Rules tab by clicking on the Title of that folder.

Note that there is no indication in the Assignments section if the Rule is applied to subfolders or not.

If you’re on a folder that has the rule assigned to it directly (e.g. it’s NOT a subfolder of a folder that has the rule assigned), you can get directly to the “Edit Content Rule” page from the Rules tab by clicking on the Title of that rule (which is always a link).

Alternately, if you’re on a folder that has the rule assigned from a folder higher up in the hierarchy, clicking on the rule Title link will take you to the folder’s Rules tab where the rule has been explicitly assigned.

If from the Rules tab, a rule is listed at active, then the assignment of that rule is being managed from a parent folder.

Creating forms without programming: PloneFormGen

Description

PloneFormGen allows you to build and maintain convenience forms through Plone edit interface.

Introduction

PloneFormGen is a Plone add-on Product that provides a generic Plone form generator using fields, widgets and validators from Archetypes. Use it to build, one-of-a-kind, web forms that save or mail form input.

To build a web form, create a form folder, then add form fields as contents. Individual fields can display and validate themselves for testing purposes. The form folder creates a form from all the contained field content objects.

Final disposition of form input is handled via plug-in action products.

Action adapters included with this release include a mailer, a save-data adapter that saves input in tab-separated format for later download, and a custom-script adapter that makes it possible to script simple actions without recourse to the Management Interface.

To make it easy to get started, newly created form folders are pre-populated to act as a simple e-mail response form.

- [PloneFormGen product page](#)

Getting started with PloneFormGen

Getting Started with PloneFormGen

Description

Learn the basics of creating web forms with PloneFormGen

Introduction

Please read general PloneFormGen information first

Getting Started

Note: The best place to start for non-technical Plone users.

To get started building a custom form using PloneFormGen, you must first create a **Form Folder**. A form folder is a lot like a normal Plone folder - you use it to organize and hold other types of content. The Form Folder also has some settings of its own that will control the behavior and appearance of the form.

Click *Add Item* and choose Form Folder from the drop-down menu.

admin ▼

☐ only in current section

Display ▼ Add new... ▼ State: Published ▼

- Collection
- Event
- File
- Folder
- Form Folder
- Image
- Link
- News Item
- Page

Plone.

this web site has just installed Plone. Do not contact the Plone Team or the Plone

do the following:

u should have a Site Setup entry in the menu in the top right corner)

y users and send out password reminders)

egistration, password policies, etc)

Create the form

Provide a *Title* and *Description* for the form. You can also change the wording of the *Submit* or *Reset* button if you wish.

Ignore the rest of this edit screen for the time being and just click *Save* at the bottom.

Add Form Folder

A folder which can contain form fields.

Default ■ Categorization Dates Ownership Settings Overrides

Title ■

Jamaican Software Developers Meetup

Description

Used in item listings and search results.

The ultimate meetup, find out what others are doing in your area

Submit Button Label

Submit

☐ Show Reset Button

Reset Button Label

Reset

Action Adapter

To make your form do something useful when submitted: add one or more form action adapters to the form folder, configure them, then return to this form and select the active ones.

☒ Mailer

Thanks Page

Pick a contained page you wish to show on a successful form submit. (If none are available, add one.) Choose none to simply display the form field values.

☐ None

☒ Thank You

☐ Force SSL connection

Check this to make the form redirect to an SSL-enabled version of itself (<https://>) if accessed via a non-SSL URL (<http://>). In order to function properly, this requires a web server that has been configured to handle the HTTPS protocol on port 443 and forward it to Zope.

Form Folder Settings

Description

Learn how to configure your form.

You now have a basic form to work with. By default, a form starts with *E-Mail Address*, *Subject* and *Comments*. These are fields you get automatically, every time you create a new form.

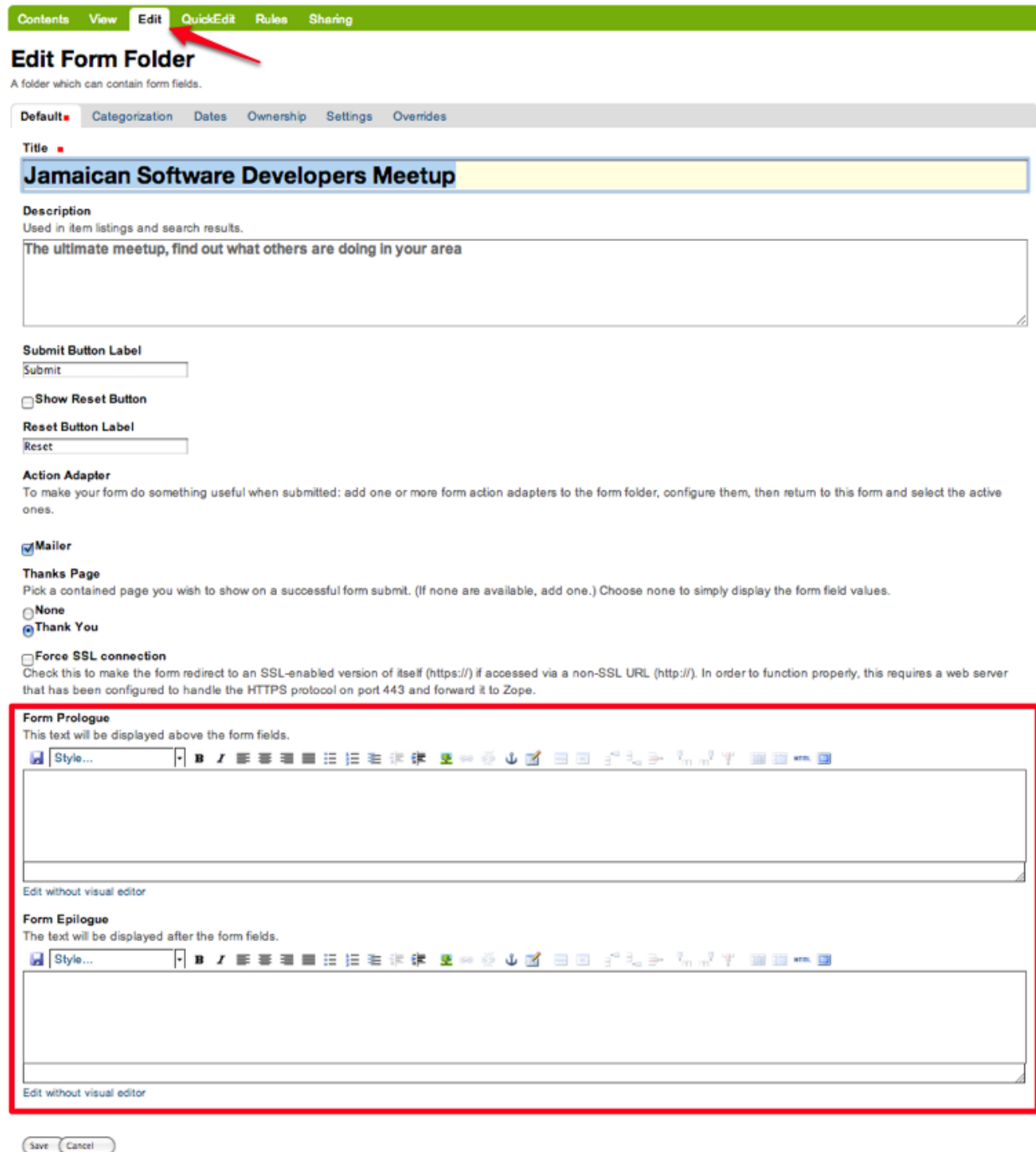
The screenshot shows a web form interface. At the top is a green taskbar with buttons: 'Contents', 'View' (active), 'Edit', 'QuickEdit', 'Rules', and 'Sharing'. On the right of the taskbar are 'Actions', 'Add new...', and 'State: Private'. Below the taskbar is a yellow status bar that says 'Info Changes saved.'. The main heading is 'Jamaican Software Developers Meetup' with the subtitle 'The ultimate meetup, find out what others are doing in your area'. The form contains three fields: 'Your E-Mail Address' (a single-line text box), 'Subject' (a single-line text box), and 'Comments' (a large multi-line text area). A 'Submit' button is located at the bottom left of the form.

Adding text to the form

Before you learn how to add new fields, or change existing ones, let's look at how you can add some simple text above and below the form.

You may want to include instructions to your site visitor about how to fill out your form, or what the purpose of the form is supposed to be. You can add that descriptive text as follows:

- Edit the Form Folder by clicking on Edit in the taskbar
- Now you'll see two Body Text areas called Form Prologue and FormEpilogue (which means before and after). Use the text editor to format your content, include links, and even pictures. Click Save when you're all done.



Contents View **Edit** QuickEdit Rules Sharing

Edit Form Folder

A folder which can contain form fields.

Default Categorization Dates Ownership Settings Overrides

Title

Jamaican Software Developers Meetup

Description

Used in item listings and search results.

The ultimate meetup, find out what others are doing in your area

Submit Button Label

Submit

☐ Show Reset Button

Reset Button Label

Reset

Action Adapter

To make your form do something useful when submitted: add one or more form action adapters to the form folder, configure them, then return to this form and select the active ones.

☒ Mailer

Thanks Page

Pick a contained page you wish to show on a successful form submit. (If none are available, add one.) Choose none to simply display the form field values.

☐ None

☒ Thank You

☐ Force SSL connection

Check this to make the form redirect to an SSL-enabled version of itself (<https://>) if accessed via a non-SSL URL (<http://>). In order to function properly, this requires a web server that has been configured to handle the HTTPS protocol on port 443 and forward it to Zope.

Form Prologue

This text will be displayed above the form fields.

Style... [Rich Text Editor]

Edit without visual editor

Form Epilogue

The text will be displayed after the form fields.

Style... [Rich Text Editor]

Edit without visual editor

Save Cancel

Overrides

When in the *Edit* mode on the Form Folder, notice the *Overrides* tab on the right-hand side of the page properties tabs. Form submission behaviors can be customized in this tab.

Read the on-screen help to give you an idea what sort of customizations are possible here. Depending on what you need to do, you may need to learn how to write some basic TALES or Python code. Don't be afraid though!

Contents View **Edit** QuickEdit Rules Sharing

Edit Form Folder

A folder which can contain form fields.

Default **Categorization** Dates Ownership Settings **Overrides**

Custom Success Action
Use this field in place of a thanks-page designation to determine final action after calling your action adapter (if you have one). You would usually use a template or script. Leave empty if unneeded. Otherwise, specify as you would a CMFFormController action type and argument, complete with type and argument (e.g. "redirect_to:string:thanks-page" would redirect to 'thanks-page').

Custom Form Action
Use this field to override the form action attribute. Specify a URL to which the form will post. This will bypass form validation, success action adapter, and success message.

Form Setup Script
A TALES expression that will be called when the form is displayed. Leave empty if unneeded. The most common use of this field is to call a python script to pre-populate request.form. Any value returned by the expression is ignored. PLEASE NOTE: errors in the evaluation of this expression will cause an error on form display.

After Validation Script
A TALES expression that will be called after the form is successfully validated, but before calling an action adapter (if any) or displaying a thanks message. Leave empty if unneeded. The most common use of this field is to call a python script to clean up form input or to script a redirect. Any value returned by the expression is ignored. PLEASE NOTE: errors in the evaluation of this expression will cause an error on form display.

Overview of Field Types

Description

Learn about the commonly used field types in PloneFormGen.

By now you've seen how to create a new form. The next step is to learn how to add new fields to the form.

There are a lot of different types of fields that one can employ. You've probably seen many of those types of fields around on the Internet like a text box, a list you can choose from, a checkbox, radio buttons, and so on.

Some of the field types that are included with PloneFormGen will not be discussed here as they are used more for advanced applications than for basic web forms. The field types discussed here should be all you need to create useful forms in Plone.

Common field types

Navigate to your Form Folder and push the Add Item button. You should see a long drop-down menu appear. There's a lot of choices, but we've chosen the four most common to focus on for this tutorial.

String Field This is probably the most commonly used field. It's a simple, one-line text box for gathering info like name, address, e-mail, phone number and so on.

Text Field A large text box for gathering things like comments or other long-form text responses.

Selection Field This field type is to gather one choice from a list of selections. The list can be checkboxes, radio buttons, or a text list.

Multi-Select Field Use this field type if you want to let your site visitor select multiple items from a list.

You'll notice that there is a special icon for each field type. It's a good idea to get familiar with how each one looks so you'll be able to recognize the different field types at a glance.

In the next section, you'll see how to add and configure a string field; the most common type of form field.

Adding a String Field

Steps to adding a single text line field called a String.

Navigate to your Form Folder and click Add Item. Select String Field from the drop-down menu.

The string field creates a simple one-line text box. Some common uses for this field type are:

- First Name
- Last Name
- Email
- Street Address
- Lots more!

Basic field info

Many field types have the same information on their edit screen. Here's an explanation of each of these:

Field label The title of the field.

Field help Some text you can provide to the form submitters to help them figure out what you're asking for.

Required Is this a required field?

Hidden Is this a hidden field? This is useful for passing data back to yourself.

Default You can supply a default value that the form submitters can change if they wish.

Other types of information can be defined for other field types, and we will cover those as they come up. Since we're talking about the string field, let's take a moment to talk about validation.

Validators

Validation is a feature common to many web forms. A validator checks that the input being provided conforms to a particular format. For example the "email address" validator simply checks that the input contains an @ sign (i.e. contact@groundwire.org). There are a set of standard validators available for string fields such as: Valid US phone number and Valid zip code.

In practice validators can be helpful, but in some cases they may be too restrictive. If your site visitors are from outside the US, they may become frustrated with trying to get past US-centric form validators!

PloneFormGen editing environment

Note that PloneFormGen has both an *Edit* tab and a *QuickEdit* tab on the taskbar. The *Edit* mode mainly allows you to edit major settings for the entire form. The *QuickEdit* mode allows you to interactively edit the form. It's much a much easier way to add, edit, delete and reorder fields.

In *QuickEdit* mode, you may add fields by dragging them from the toolkit at the right side of the page and dropping them into the target position in the form.

If you want to see the form the way that your site visitors would, you can click on View in the taskbar. Likewise you can get to the editing environment from the normal view, by clicking the *QuickEdit* tab in the taskbar.

Note that you cannot complete or submit the form when working in the editing environment. You will need to return to the normal form view first.

Changing the Order of the Fields

Description

How to rearrange fields in your form.

By now you're probably wondering how to change the order that fields appear on your form. By default, when you create a new field, it will appear at the bottom of your form. Often, this isn't the result you wanted to see.

Navigate to your Form Folder and click on the *QuickEdit* tab on the taskbar to enter the form editing environment.

Notice that the column on the far left called Order. You can move the position of each field by clicking and dragging each field around if you hover your mouse pointer in the Order column. Continue to rearrange fields until you have the layout you want.

To see the results, you need to click on the View tab to see your form the way your site visitors will.

Text Field

Description

How to add a text box for collecting comments or other text.

A *Text field* is like a string field except that it's a large box instead of a single line. You can change the size of the field by adjusting the Rows and Max Length parameters.

Rows controls the vertical height of the text box. Max Length controls the number of characters the site visitor can input at a time. If you want to limit a Comments text box to short comments, you might choose 500 characters as the limit.

The *Default* field can be used if you want to suggest some content to the site visitor or wish to show an example of the sort of information you want from them.

You can also *Reject Text with Links* to discourage people from adding links to your text area field. One reason why you might choose to do this would be if you're asking someone to enter some information and you don't want them to simply link to a blog entry or other online content.

Selection and Multi Select

Description

How to create menus and checkboxes for selecting items from a list within your form.

In addition to text boxes and the string field, *selection* and *multi-select* are commonly used field types. They allow site visitors to choose a selection from a list, drop-down menu, radio buttons or checkboxes.

Selection field

As before, you must be in your Form Folder to add additional fields. Choose *Selection Field* from the list in the Add New drop-down menu or the QuickEdit toolkit.

In addition to the regulars like *Field Label* and *Field Help*, there are *Options* and *Presentation Widget* to consider.

Options

Options is for establishing what the options in the field are going to be. Each option should be separated with a line break.

Presentation widget

The *Presentation Widget* is the kind of graphic used for gathering the input: you can choose either Radio Button or Selection List (a drop-down menu). The option Flexible simply means that you leave it up to PloneFormGen to decide which widget is most appropriate. Basically, if you have more than three choices it will use the selection list. Less than three will appear as radio buttons.

Value|label

Entering Options can be done in one of two ways. Either "one line per option" or the "Value|Label format". One line per option is described above, but what about *Value|Label*?

Let's say you want to present some choices, but the actual value recorded by the form is different than what the form submitter sees on the screen. For example, let's say that you want to ask a site visitor what county they live in, but in your program work you classify counties into regions like "Northwest" and "Southeast". Instead of asking the visitor to identify which region they live in (which some might get wrong or misunderstand) you could present them a list of counties.

In the above example you would format Options like this:

```
Northwest | Jefferson
Northwest | Island
Northwest | Mason
Southeast | Columbia
Southeast | Asotin
Etc . . .
```

In this example, if a visitor selects “Jefferson” as their county, the form would record the entry as “Northwest”.

Multi-select

Multi-select is very similar to a Selection Field except that you have an additional widget (the checkbox) and your site visitors can choose many options from one list.

If you plan to use the Selection List widget for a multi-select field, it is helpful to include a note about the Control key in Field Help. In order to actually choose more than one option in the list, you must hold down the control key (CTRL) on your keyboard and then click to select/de-select options. Because of this, it is most often the best choice to use the Checkbox widget instead of Selection List for a multi-select field.

The Thank You Page

Description

How to customize the page a site visitor sees after having submitted the form.

After a site visitor has filled out your form and clicks submit, they will see a page thanking them for their input. Look in your Form Folder and you should already see a Thank You page in the contents.

It doesn’t matter where the Thank You page appears in your Form Folder contents. It will always work the same, no matter its folder position.

By default, the thank you page only lists a summary of what the site visitor put into the form. Here’s an example of what that looks like, to the right.

Note: You will only see this result if you fill out the form and submit it. Otherwise, if you try to navigate or link to the thank you page directly, you get a message that says something like “no input was received”.

Add content to the thank you page

The above example is decent, but there will be times that you may want to say something more, or even provide a few links for your site visitor to follow. Edit the Thank You page in your list of Form Folder contents. You should now see the familiar Title and Description fields, but you’ll also see:

- Thanks Prologue
- Thanks Epilogue
- No Submit Message

If you’ve chosen to display any field results the Prologue content will appear before those results and the Epilogue content will appear after. If you aren’t going to display field results, just use the Prologue.

Should you wish to change the No Submit Message (remember, if you just hit submit on your form without filling out any fields, you’ll see this message) you can do so here. You have the full power of Plone’s text editor so feel free to include links and formatted text here.

Fields

thanks-fields.gif

When editing the Thank You page, notice the Fields tab next to Default. Here's where you can control which field results to display or which ones to take out. Simply uncheck Show All Fields if you don't want to display any results on the Thank You Page.

The Mailer Adapter

Description

Learn to configure the adapter which sends an email after the form is submitted.

Overview

Adapters control what happens to the form data that your site visitors submit with the form. The Mailer Adapter will send form data to an email address or addresses of your choosing. The Save Data Adapter will save the results in your Form Folder so that you can export them any time you wish. You can even use adapters concurrently to get the functionality of both.

Configuring the mailer adapter

The *Mailer Adapter* is probably the more complex of the two adapters covered in this tutorial. As such, we won't go through and explain all the options present in the Mailer Adapter. However, we will cover the most important options available.

Go ahead and edit the Mailer Adapter in the form editing environment. The first screen you see look like this:

From here you can do three things:

- Change the Title of your Mailer Adapter (really no reason to, unless you have more than one)
- Choose a recipient name
- Choose a recipient email address

Addressing

Now notice the tabs [default] [addressing] and so on. Click on [addressing].

The *Addressing* screen allows you to make selections about other recipients and dynamically populate the *From* and *Reply-to* fields directly from form data.

Message

The Message tab allows you to configure the:

- Subject Line
- Email body content

- Form field data that appears in the email message

Template, encryption, & overrides

These options are beyond the scope of this tutorial as they require a discussion of HTML, TALES and Python programming languages as well as an understanding of mail server configuration.

The Save Data Adapter

Description

Collect and save answers from each form submission with the Save Data Adapter.

Unlike the Mailer Adapter, the *Save Data Adapter* isn't automatically created when you build a new Form Folder. As such, you must add it yourself by clicking Add New (when you're in your Form Folder) and selecting the *Save Data Adapter* from the drop-down menu. Or, drag and drop it from the toolbox in QuickEdit mode.

The first thing you'll see is a screen like the one to the right.

Configure the adapter

Enter in a Title that sounds good to you (*My Saved Form Data*, for example). Your site visitors won't see this title at all, so anything will do.

Next, you can select some *Extra Data* to store if you wish. Most of these choices are for fairly esoteric things, but Posting Date/Time could be helpful if you want to know when someone filled out your form.

Download Format depends on what your preferences are, but *Comma-Separated Values* is probably the most common. It really depends on if you are going to plug the info into a database, and what type of file is most appropriate for that. If the info is for human eyes only, then it doesn't really matter what you pick here. Europeans: you may choose to use a colon rather than a comma for CSV on the PloneFormGen config panel in site setup.

Now click on *Save* to finish. You do not need to put anything into the Saved Form Input box. If your Save Data Adapter had any data in it, it would appear in that box.

That's all there is to it. Sit back and wait for the data to come pouring in!

Retrieving your data

Once your form has been filled out a few times by site visitors, you can start retrieving the form data. You need to navigate to your Form Folder and click on the Contentstab to get a view of all contents.

Now click on your *Save Data Adapter*. You should then see a screen like this one at the right.

You can download the form data as many times as you want. The data will always be there as long as you don't delete the *Save Data Adapter*. If you click *Clear Saved Input* all data will be erased as well.

Safety Net

A common way of saving important data is to use two *save data* adapters in your form. Use one to occasionally harvest new data and clear it. Use the other to save all your history.

Note: This document was originally written by Sam Knox for Groundwire. Thanks to both Sam and Groundwire for passing it on to the Plone Foundation.

PloneFormGen topics

Installing PloneFormGen

Description

PloneFormGen is a Plone add-on product, and is not included with Plone. Fortunately, it's easy to install.

PFG installs just like most other Plone add ons. Edit the `buildout.cfg` file at the top of your Plone instance and look for the `eggs =` section that specified Python Packages that you wish to include. Add `PloneFormGen`:

```
eggs =
    Plone
    ...
    Products.PloneFormGen
```

Run `bin/buildout` and restart your Plone instance. Dependencies will be loaded automatically.

After restarting Plone, visit your site-setup page and use the “add on” configuration page to activate PloneFormGen.

Overriding field defaults dynamically

Description

PloneFormGen allows you to supply dynamic field defaults by specifying a TALEs expression in the Default Expression field of the overrides fieldset (sub-form). This how-to explains what that means, and offers a few examples.

Creating a dynamic field default means to have a form field's initial value change with context. We might, for example, wish to use a member's e-mail address, which would vary with the user. Or, we might be looking up some data via a catalog or RDBMS query, and wish to supply that to the user for correction.

Template Attribute Language Expression Syntax (TALES) is a simple notation that allows determination of a value via path (as in `path/to/object`), string or Python expressions. It is used in *Zope's Template Attribute Language* (TAL), and is ubiquitous in Plone templates. This how-to does not teach you TALEs; for that, try the [Zope Page Templates Reference](#).

Warning: Please note that it's easy to make a mistake when working with TALEs fields that will cause an error when you try to display your form. Stay calm! Take note of the error message, and return to the field edit form to fix it. Don't be scared of this kind of error.

Example

Let's say you wish to put the member's id in a string field default. You may do that with the TALEs expression:


```
member/id
```

This is a path expression. “id” is found in the “member” object and returned.

There’s a gotcha here. What if the form is viewed by an anonymous visitor? They’ll receive an error message. We can avoid that with the expression:

```
member/id | nothing
```

The vertical bar (|) marks alternate expression that is used if the left-hand expression is empty or can’t be evaluated. Here we’re saying to show nothing if member/id can’t be evaluated. Using Python

You may also use Python expressions:

```
python: 5 + 3
```

would supply a value of 8. This is trivial, but what about:

```
python: DateTime() + 7
```

This would supply the current date/time plus seven days.

The name space

Here are the objects available when your expression is evaluated.

TALES context

here The current object. A bit dangerous since this varies depending on context.

folder This will be your form folder.

portal The portal object.

request The REQUEST object. Note that request/form contains form input.

member The authenticated user’s member data – if any.

nothing Equivalent to Python None.

folder_url URL of the form folder.

portal_url URL of the site.

modules Module importer.

Note: Some of these identifiers are supplied by PloneFormGen and are not available in other contexts.

When you compose your TALES expression, keep in mind that it will need to return different types of data for different types of fields. For simple field defaults, return a string value; for the lines field, return a list or tuple.

Calling a Python script

You’ll be frustrated fast if you try to do anything smart in a single TALES expression. If you need to do something more complicated, add a Python Script to your form folder and call it via TALES. For example, if you added a script with the id getEmail, you could call it with the expression:

```
folder/getEmail
```

Make sure your script returns the value you wish to use as a field default, in the appropriate format.

Setting Many Defaults With One Script

If you need to dynamically set several fields, you may do it with one script. Call the script by specifying it in the Form Setup Script field of the form folder's overrides fieldset.

Set the form fields by putting them in the request/form dictionary. Make sure you don't overwrite anything that's already in the dictionary, as that is probably previously submitted input.

For example, we could create a Python Script (using the Management Interface) in the form folder:

```
request = container.REQUEST  
  
request.form.setdefault('topic', 'value from python script')
```

If the script id was setTopicDefault, we'd call it by putting:

```
here/setTopicDefault
```

in the *Form Setup Script* field of the form folder's overrides fieldset.

Creating custom validators

Description

PloneFormGen allows you to create a custom field-input validator by specifying a TALEs expression that will be used to validate input when it's submitted. This how-to explains what that means, and offers a few examples.

Template Attribute Language Expression Syntax (TALES) is a simple notation that allows determination of a value via path (as in path/to/object), string or Python expressions. It is used in *Zope's Template Attribute Language* (TAL), and is ubiquitous in Plone templates. This how-to does not teach you TALEs; for that, try the [Zope Page Templates Reference](#).

Warning: Please note that it's easy to make a mistake when working with TALEs fields that will cause an error when you try to display your form. Take note of the error message, and return to the field edit form to fix it. Don't be scared of this kind of error.

The rules for writing a validator are:

- You should validate against the the variable value, which will contain the field input. Note that – for simple fields – value will be a string. But, for a lines field, the contents of value will be a list.
- Return False or zero if you wish to accept the input.
- Return a string containing a user-feedback message if you don't wish to accept the input.
- Don't change the value variable. It won't do you any good.

Example

Let's say that you are operating a restaurant that serves only dishes containing spam. You may wish to check to make sure that the input to a string or text field contains "spam". You may do that with by setting a custom validator that reads:

```
python: 'spam' not in value and 'Input must include spam.'
```

The odd logic comes from the need to return *False* for valid input. Look at a couple of examples of validation in action with literal strings. Remember, we want to force spam on the user:

```
>>> 'spam' not in "eggs, eggs, bacon" and 'Input must include spam.'
'Input must include spam.'

>>> 'spam' not in "eggs, eggs, bacon and spam" and 'Input must include spam.'
False
```

The name space

Here are the objects available when your expression is evaluated.

TALES context

value The field input.

here The current object. A bit dangerous since this varies depending on context.

folder This will be your form folder.

portal The portal object.

request The REQUEST object. Note that request/form contains form input.

member The authenticated user's member data – if any.

nothing Equivalent to Python None.

folder_url URL of the form folder.

portal_url URL of the site.

modules Module importer.

Note: Some of these identifiers are supplied by PloneFormGen and are not available in other contexts.

Using a Python script

You'll be quickly frustrated if you try to do anything smart in a single TALEs expression. If you need to do something more complicated, add a Python Script to your form folder and call it via TALEs. For example, if you added a script with the id includesSpam, you could call it with the expression:

```
python: folder.includesSpam(value)
```

Make sure your script returns False if you wish to accept the input, or an error string otherwise.

Here's what a validator script to check for spam might look like:

```
if 'spam' in value.lower():
    return False
else:
    return "'%s' doesn't seem to have spam. Try again." % value
```

Make sure your script parameter list includes value. (Alternatively, you may check the request.form dictionary, which will include form input.)

Note: Python scripts are not the same as the Custom Script Adapter. The latter is meant to make it easy to add a custom adapter that's processed in the same way as the mail or save-data adapter. Python scripts Python code fragments that act like functions. They are added via the Management Interface.

Using a selection field to pick mail destination

Description

You may allow form users to use a selection field to choose a destination address for their form input.

I'm trying to use a PloneFormGen form as a support center for my project and I would like to have the mail sent to different email addresses based on a choice from a selection field.

How can I do it?

The form

First, create a selection field in your form

In the Options field, specify your set of possible destination addresses in a “valuellabel” format where the e-mail address is the value and its readable name the label. For example:

```
softwarehelp@example.org|Software Support Desk
hardwarehelp@example.org|Hardware Support Desk
```

Then, pick the address (the actual e-mail address value, not the label) you wish selected by default. Put it in the Default field. Make sure the Required checkbox is selected.

Save the form field.

Configuring the mailer

Now, edit the mail adapter for your form. (Navigate to the form folder, click on contents, find your mail adapter and follow the link; select the edit tab.)

Choose the *[addressing]* sub-form and find the Extract Recipient From field. You should see a None choice and a list of all of the selection fields in your form. Select the field you just created and save your changes.

Restyle a form

Description

How to inject CSS into a form page to turn a label green ... or pretty much anything else.

The general answer to “how do I restyle a form” questions is: use CSS.

The underlying Archetypes form generator surrounds every form element with a `<div>` with a distinct ID. For example, a sample form with a textarea contents field has the generated XHTML:

```
<div class="field ArchetypesTextAreaWidget"
  id="archetypes-fieldname-comments">
  <span></span>
  <label for="comments">Comments</label>
  <span class="fieldRequired" title="Required">
    (Required)
  </span>
  <div class="formHelp" id="comments_help"></div>
  <textarea rows="5" name="comments" cols="40" id="comments"></textarea>
  <input type="hidden" name="comments_text_format" value="text/plain" />
</div>
```

That’s more than enough ID and Class selectors to do pretty much anything in the way of visual formatting.

How do we get the CSS into the form’s page? You could add it to the site’s css, but there’s a much easier way. Using the Management Interface, create an object of type File inside your form folder. Set its Content Type to “text/plain” and give it the ID “newstyle”.

Let’s turn the label for the comments field green. Just fill in the big text field on your file with:

```
<style>
#archetypes-fieldname-comments label {
  color: green;
}
</style>
```

Now, save it, return to the Plone UI and edit your form folder. Specify “here/newstyle” for the Header Injection field of the overrides pane. Now, enjoy your green label.

Putting checkboxes in a row

Now, for a more useful example. It’s a common requirement to want to put a set of checkbox fields on a single line.

The easiest way to set this up is to create the list of checkboxes as a multi-selection field with “checkboxes” designated for display. That’s going to generate markup that will look something like this:

```
<div id="my-questions">
  <div class="formQuestion label">
    My Questions
    <span id="my-questions_help" class="formHelp"></span>
  </div>
  <div id="archetypes-value-my-questions_1" class="ArchetypesMultiSelectionValue">
    <input type="checkbox" name="my-questions:list" value="a" id="my-questions_1"
    ↪class="blurrable">
    <label for="my-questions_1">Choice A</label>
```

```
</div>
<div id="archetypes-value-my-questions_2" class="ArchetypesMultiSelectionValue">
  <input type="checkbox" name="my-questions:list" value="b" id="my-questions_2"
↪class="blurrable">
  <label for="my-questions_2">Choice B</label>
</div>
<div id="archetypes-value-my-questions_3" class="ArchetypesMultiSelectionValue">
  <input type="checkbox" name="my-questions:list" value="c" id="my-questions_3"
↪class="blurrable">
  <label for="my-questions_3">Choice C</label>
</div>
</div>
```

Note that each checkbox/label pair is in a DIV with the class “ArchetypesMultiSelectionValue”. The basic CSS couldn’t be simpler:

```
<style>
#my-questions div.ArchetypesMultiSelectionValue {
  float: left;
}
</style>
```

Of course, you’ll need to do some more styling. First of all, you’ll need to set a `clear: left` on the following control. And, you’ll need to do some padding.

Putting string fields in a row

Making string fields display horizontally is a little different than the solution for checkboxes. There is no div wrapping the string fields like there is with check boxes. To get around this, add a fieldset and put the fields in the fieldset. This also lets you isolate the horizontal fields from other vertical fields. Here, two fields are required, one is not. The markup will look similar to this:

```
<fieldset class="PFGFieldsetWidget" id="pfg-fieldsetname-name">
  <div class="formHelp" id="name_help"></div>
  <div class="field ArchetypesStringWidget " id="archetypes-fieldname-first-name">
↪ <span></span>
    <label class="formQuestion" for="first-name"> First Name <span class=
↪"required" title="Required" style="color: #f00;"> &#x25a0; </span> </label>
    <div class="formHelp" id="first-name_help"></div>
    <div class="fieldErrorBox"></div>
    <input type="text" name="first-name" class="blurrable firstToFocus" id=
↪"first-name" size="20" maxlength="30" />
  </div>
  <div class="field ArchetypesStringWidget " id="archetypes-fieldname-middle-
↪initial"> <span></span>
    <label class="formQuestion" for="middle-initial"> Middle Initial </label>
    <div class="formHelp" id="middle-initial_help"></div>
    <div class="fieldErrorBox"></div>
    <input type="text" name="middle-initial" class="blurrable firstToFocus"
↪id="middle-initial" size="1" maxlength="1" />
  </div>
  <div class="field ArchetypesStringWidget " id="archetypes-fieldname-last-name">
↪<span></span>
    <label class="formQuestion" for="last-name"> Last Name <span class=
↪"required" title="Required" style="color: #f00;"> &#x25a0; </span> </label>
    <div class="formHelp" id="last-name_help"></div>
```

```

        <div class="fieldErrorBox"></div>
        <input type="text" name="last-name" class="blurrable firstToFocus" id=
↪ "last-name" size="30" maxlength="255" />
    </div>
</fieldset>

```

Here is the CSS:

```

<style>
/* Displays the 3 string fields horizontally. Turn off the clear from Public.css. ↪
↪ This is necessary to display horizontally. */
#pfg-fieldsetname-name div.ArchetypesStringWidget {
    float: left;
    clear: none;
}

/* needed for space between fields */
#archetypes-fieldname-middle-initial
{
    padding: 0 1em;
}

#content fieldset#pfg-fieldsetname-name
{
    /*Hide the border on the fieldset */
    border-style: none;
    /*Need this to left align the fields inside the fieldset with the fields outside the ↪
↪ fieldset*/
    padding-left: 0;
}
</style>

```

An alternative way to inject CSS

Let's say you've got a lot of CSS. You may want to use an external style sheet file rather than inject the whole bundle into the header with every form display.

Let's say the CSS resource is named `form_styles.css`. Then, just put the following in your overrides / header injection field:

```
string:<style>@import url(form_styles.css)</style>
```

We can get a little fancier to generate absolute URLs for the style file:

```
string:<style>@import url(${here/form_styles.css/absolute_url})</style>
```

using the string interpolation feature of TALES.

Note: Need to do something more sophisticated? You can use a Python script to generate dynamic CSS or JavaScript.

Adding a JavaScript event handler to a form

Description

Need to make your PFG forms more dynamic? It's easy to add JavaScript.

There are two basic steps to injecting JavaScript into a PFG form:

1. Use the Management Interface to create a text file (object type: file; mimetype: text/plain) either inside the form folder or in a skin folder;
2. Use the form folder's edit / overrides pane, header injection field to tell PFG to inject it into the form.

Injection

Let's look at the second step first. Let's say that your JavaScript file is named `form_js`. Then just specify:

```
here/form_js
```

in the header injections override field.

JavaScript

There are a couple of considerations here:

1. Since this is a header injection, you'll need to supply the `SCRIPT` tags;
2. You'll nearly certainly want to use jQuery to attach the event handler, since jQuery is part of Plone.

```
<script>
jQuery(function($) {
    $('#my-questions :input')
        .click(function() {
            alert('checkbox clicked');
        });
});
</script>
```

This code fragment shows off both, and attaches an alert to every input in the `my-questions` field.

Note the use of the common jQuery idiom:

```
jQuery(function($) {
    ...
});
```

This accomplishes a couple of things:

1. it sets the code up to run once the page is loaded;
2. it aliases "jQuery" to "\$" so that we may use common jQuery shorthand.

An alternative injection

If you want to use a separate JavaScript file that is actual JS (no `script` tags) and will be shared among forms, use the header injection override like this:


```
string:<script src="form_scripts.js" />
```

assuming your script is named `form_scripts.js`. You may make it a little more sophisticated if you need an absolute URL:

```
string:<script src="{here/form_scripts.js/absolute_url}" />
```

using TALES string interpolation.

Customizing an individual thanks page

Description

It's not hard to customize the thanks page for an individual form. This trick is particularly useful for purposes like adding 'pay now' buttons.

If you can tolerate a little work in the Management Interface, you'll find it very easy to customize the Thanks Page for an individual form.

The steps:

1. Create your form;
2. Bring up the Management Interface; navigate to `portal_skins/PloneFormGen`;
3. Open the `fp_thankspage_view` template; push the Customize button; this puts an editable copy of the thanks page template in your custom skin folder.
4. Step back to the Custom folder listing (still in the Management Interface); cut the `fp_thankspage_view` template;
5. Navigate to your form folder; paste it there.
6. Edit the template to insert your Pay/Donate Now form and button code, or whatever other custom code you might need just for this form.

Note: Note: If there is already an `fp_thankspage_view` template in your custom skin folder (perhaps because you've already customized the template for the site), you'll be cutting and pasting a new copy.

What's in a Request

Description

If you're trying to use `PloneFormGen` overrides, you're going to need to use the request object. Here's a quick trick for exploring it.

As a page is assembled by Zope and Plone following a browser request, information about that request is bundled into a non-persistent, pseudo-global request object. This object is available in the scripts, templates and TALES expressions you may use in creating `PloneFormGen` overrides. It will contain the form input submitted by the user.

To effectively write more complex overrides, you're going to need to know how to get information out of the request object.

Note: The Request class itself is well-documented in the Zope help system (API section) and in the source at Products.OFSP-2.13.2-py2.X/Products/OFSP/help/Request.py.

Here's a quick recipe that will help you examine the form input contained in the request.

- Jump into the Management Interface and navigate to your PFG Form Folder. Inside it, create a Page Template named showrequest. Now, just before `</body>`, add:

```
<div tal:replace="structure request" />
```

Note: when the request object is called, it renders a readable, HTML version of the data. We use “structure” to prevent escaping the HTML.

- Give your template a title and save it away.
- Return to Plone and your form folder. Edit it, and on the form's [overrides] pane, set a Custom Success Action of:

```
traverse_to:string:showrequest
```

Note that this will override any thanks page you've specified. Just clear it when you're done developing.

Now, just fill out your form, and submit it. You should see the contents of the request object. Take a particular look at the form section. That's a dictionary available as `request.form` when you're composing an override.

Creating a Multi-Page Form

Description

You can create a multi-page form as a chain of form folders.

Creating a multi-page form is probably much simpler than you might suppose. You may do it by just creating a sequence of form folders that each link to the next. The basic procedure is:

- Create your sequence of form folders, typically all in the same normal Plone folder;
- On all but the last form folder, turn off all action adapters and set the Custom Success Action override to “traverse_to:string:id-of-next-form-folder”;
- On all but the last form folder, set the Submit Button Label to something like “Next” and turn off the cancel button.
- On all but the first form folder, set the Exclude From Navigation flag in the properties tab;
- In each form folder, create a set of hidden form fields matching all the fields in all the previous forms;
- In the last form, turn on your real action adapter(s).

As your user moves from form page to page, input will be automatically saved in the hidden fields of subsequent pages.

Note: A Note on Hidden Fields: The hidden flag is not available for all form field types, but you don't need it. String, Text and Lines fields are adequate to carry all the basic data. Use a hidden Lines field to hold multiple selection field input, string or text for the rest.

An added bonus

If you want to create a sequence of forms, where the answers on form_A could lead to a form_B or form_C, you can use a `traverse_to` to override.

- Create a selection field in form_A, which could be called ‘formnext’;
- As values in the selection field, put the paths to the next forms in the sequence;
- Then, in the form overrides -> custom success action use

`traverse_to:request/form/formnext`

Note: use `traverse_to` as opposed to a `redirect_to` as this will preserve the form object in the request.

Adding CAPTCHA Support

Description

PloneFormGen has built-in support for Re-Captcha. This how-to tells you how to enable it.

PloneFormGen and CAPTCHA Fields

When PFG is installed in a Plone instance via add/remove products, it will look for evidence that either `collective.captcha` or `collective.recaptcha` are available. If that’s found, the CAPTCHA Field will be added to the available field list.

If you are using `collective.recaptcha`, you need to take the additional step of setting your public/private keypair. You get these by setting up an account at [recaptcha.net](https://www.recaptcha.net). The account is free. You may specify your keypair in the PFG configlet in your site settings.

Note: If you add a captcha facility *after* installing PFG, you will need to reinstall PFG (via add/remove products) to enable captcha support.

Installing collective.recaptcha

Add the following code to your `buildout.cfg` to install `collective.recaptcha` and `Products.PloneformGen` together:

```
[buildout]
...
eggs =
    Plone
    ...
    collective.recaptcha
    Products.PloneFormGen
    ...
```

- Re-run `bin/buildout` and relaunch your `zope/plone` instance.
- Open the PortalQuickinstaller or plone control panel and install (or reinstall if already installed) `PloneFormGen`.

- Open the PloneFormGen configlet in the Plone control Panel and fill in the fields with your Public and Private Keys of your ReCaptcha Account. Obtain keys from [reCaptcha.net](https://www.google.com/recaptcha/).

Frequently Asked Questions

Q. How can I make a date/time field default to current time?

In the field's "overrides" fieldset, specify as Default Expression:

```
python:DateTime()
```

Note that you may do some simple date arithmetic. To set the default a week after server time, use:

```
python:DateTime() + 7
```

Q. I've made an error in a TALES expression, and now I can't view or edit my form!

An error in a TALES override may prevent you from viewing the form, but it shouldn't stop you from editing it.

To edit, navigate to the form (you'll see your error).

- If the error is in a form override, add "/atct_edit" to the end of the URL to reach the editor. That will allow you to reach the form editor; now go to the overrides fieldset and fix the problem.
- If the error is in a field override, add "/folder_contents" to the end of the URL to reach the folder contents. Click on the troubled field; you'll again get an error. Now, add "/atct_edit" to the end of the URL to reach the editor.

Q. How do I make a field default to the member's name/address/id?

In the field's override fieldset, set the Default Expression to:

```
here/memberEmail
```

memberEmail is a method of the form folder which will return a member's e-mail address, if one is available, and an empty string otherwise.

You may also use "here/memberFullName" to get the member's name, and "here/memberId" to get the login id.

Note: memberEmail, memberFullName and memberId are a convenience facility of PloneFormGen. They are not part of the Plone API.

Q. Where is the encryption option?

I understood PFG could GPG encrypt mail, but can't find the option to do it.

Navigate to your mail adapter and edit it. Look in the fieldset list (the list of bracketed sub-forms at the top of the form). Do you see an encryption field set title? If so, you've found the option. If not, it means that PFG was unable to find the gpg binary when it started. Read the README_GPG.txt file in the PFG product folder for details on how to solve this problem.

Don't forget that after you install GnuPG, you'll need to restart Zope or refresh your PFG product. Where is the save-data adapter?

Q. I've added a form folder, and the action adapter list includes "None" and "Mailer". Where is the save-data adapter?

You need to add it to the folder via the add-item drop-down.

A mailer adapter is in the "sample" form created when you add a form folder because it's probably the most common use. Other adapters need to be added.

Q. Why are these action adapters content types? Why aren't they built into the form folder?

There are several reasons. One is that doing it this way makes it easy to copy configured action adapters from one form to another.

Q. When I attempt to submit a form, I get an AssertionError "You must specify a recipient e-mail address in the mail adapter."

The error is occurring because PloneFormGen doesn't have a recipient address to which to mail the form input.

To fix this, choose the contents tab of your PFG form folder. Navigate to the mailer and use its edit tab. Choose the "addressing" fieldset and specify a recipient address.

By the way, if the recipient address isn't specified, PFG tries to use the e-mail address of the form folder's owner. You'll see this error if you've failed to set an e-mail address in personal preferences.

PloneFormGen missing from Add list?

I installed the release of PloneFormGen in my Products directory in Plone 2.5.x, and neither the Management Interface (/Control_Panel/Products) nor Plone (Quick Installer) seemed to recognize it after restarting my Zope.

Zope has probably encountered an error in the course of loading the product.

Try checking your event.log for related error messages. You may wish to try starting Zope in foreground mode (bin/zopectl fg for a standalone zope) for more diagnostics.

How do I add a hidden field with the username?

Create a string field and mark it hidden.

On the overrides tab, set "here/memberId" for the Default Expression.

Note: To follow this recipe, you'll need to have permission to edit TALES fields.

Q. Dynamically populate selection fields?

Can I dynamically populate selection and/or multi-selection fields in PloneFormGen?

Yes, use the [overrides] panel of the field's edit view to set an Options Vocabulary.

It should be a TALES expression that evaluates as a list of value/label lists (tuples are also OK).

For example, let's say that we wanted a selection field populated with option values '1', '2', '3', '4' and matching visible labels 'one', 'two', 'three', 'four'. The TALES code for this would be:

```
python: (('1', 'one'), ('2', 'two'), ('3', 'three'), ('4', 'four'))
```

It’s unlikely, though, that you’ll be able to do what you need in a single line of TALES. A more typical use would be to create a python script that returns a sequence of value/label sequences. If you put that script in your form folder, you can fill in:

```
here/myscriptid
```

in your Options Vocabulary field.

Q. Could a selection field in a FormFolder be used to redirect?

I have created a custom FormFolder, using PloneFormGen. Within the FormFolder, I have created a page and added a selection field with value/label pairs equivalent to: path (url) | company department → i.e. <http://my.site/reports/accounting>|Accounting I am wondering if it is possible to create an action override that would ‘redirect_to’ the ‘selected’ value in the selection field, something like: ‘redirect_to:string: ‘ If so, how might I access the value from the selection field?

For the redirection, just put something like:

```
redirect_to: request/form/my-selection-field
```

in the Custom Success Action field on the form folder’s [override] panel.

If you need to do something more complicated, you can use the “Custom Script Adapter” in the 1.1 alpha and end your code with:

```
request.response.redirect(request.form['my-selection-field'])
```

Use a “From” address other than the site address?

One stock-field is called replyto and contains a valid email address. I want this address to be in the From: line - not just in Reply-To:. I could fill in a TALES expression to overwrite the default sender-address. But what’s the correct TALES expression for that?

By default, PloneFormGen’s mailer sends mail with the “From” address set to the site’s global “From” address (specified in site setup / Portal Settings). That’s the standard return address for portal-generated mail, but you may wish to use another.

In the mailer’s overrides sub-form, set the Sender Expression to:

```
request/form/replyto
```

to use the address filled in for the “replyto” form field.

You could also specify a literal:

```
string:test@mysite.org
```

Be cautious about using user-submitted addresses for the “From” address. It’s important that the “From” address be a real one, owned by a responsible person.

Q. Can I integrate my favorite field/widget?

I'd like to integrate a new field/widget into PloneFormGen so that it will be useful as a form field in a PFG form.

PFG is designed to allow this, but it's going to take some programming by you or the field developer. See the PFG “examples” directory for a heavily commented, really working, example of integrating a third-party field into PloneFormGen without touching the PFG or field code.

Q. Captcha field is not accessible?

Or, not always readable for some people with low vision, or when using mobile phones this type of control is strongly blocking

To effectively replace a Captcha, just add a mandatory text field (must match the size of two chars. max.) That can be called e.g. ‘Filter’ as help text with the following question: “to avoid spam can you answer this question: 7+2-1 = ?

Next, modify the object and choose the menu ‘overrides’ and fill in the “custom validator” by this expression:

```
python: value != '8' and 'the answer is false'
```

Advanced topics

Simple SQL CRUD with PloneFormGen

Description

A step-by-step lesson in using PloneFormGen to read, insert and update rows in a single SQL table.

Introduction

One of the goals of PloneFormGen is that it should be useful for simple update operations on an external database.

This tutorial covers the use of PloneFormGen to update and insert rows in a single-table SQL database.

The simple application we'll develop here would need quite a bit of polishing before you'd wish to expose it to the public, but it will demonstrate the basic techniques.

Skills required to understand this tutorial include:

- The ability to add an SQL database connection and Z SQL Methods via the Management Interface and to understand what they do. If you've never read the Relational Database Connectivity chapter of the Zope Book, take some time now to do it; it's fundamental.
- Simple Python scripting via the Management Interface. Read the python-based scripts portions of the Advanced Zope Scripting chapter of the Zope Book if you're new to this vital Zope/Plone development skill.

Our basic steps will be to:

- Add a database and table to our SQL database and create a matching form in PloneFormGen;
- Add a Z SQL Method to insert rows into the database and show how it can be used in PFG;
- Add a Z SQL Method to read a row, write a Python script wrapper and use it to fill out the fields of our form;

- Add a Z SQL Method to update a row, write a Python wrapper for it and the insert method and use it as a form action;
- Consider the security implications of the fact that the SQL access methods we just created are not part of the Plone workflow.

By the way, we'll be skipping the "D" in CRUD. Deletion is up to you. :)

Note: This tutorial uses *Z SQL Methods* because they're easy to teach quickly. If you're doing any significant database work with any Python application, [SQLAlchemy](#) is a much more scalable way to use a relational database from Plone.

Database table & form

In this step, we create a simple database table and a matching form.

The database

Hope you're not feeling too ambitious at the moment, because this is going to be a demonstration table. It's going to have three columns:

uid A unique ID that's the primary key for the table. We'll make it auto-increment so that our SQL server (MySQL in this case) will do the work of keeping it unique.

string1 A simple string.

string2 Another simple string.

(I told you this was simple!)

Create a test database and then the table. In MySQL, the CREATE code to set up the table is:

```
CREATE TABLE simple_db (
  uid bigint(20) unsigned NOT NULL auto_increment,
  string1 varchar(255) NOT NULL default '',
  string2 varchar(255) NOT NULL default '',
  PRIMARY KEY (uid)
) TYPE=MyISAM;
```

Now, set up an SQL user with privileges adequate to select, insert and update the table. Use that user identity to set up an SQL database connection object via the Management Interface. (If you're using MySQL, this would be a Z MySQL Database Connection.) The connection must be in a place where it will be visible to the form you'll be creating.

The form

Create a PloneFormGen form with three fields:

uid An string field with id uid, marked hidden, with a default value of "-1". Later in the tutorial, we'll use the "-1" as a marker value to indicate a new record.

string1 A string field with id string1.

string2 A string field with id string2.

Note that the form field ids must exactly match our column ids. You can script your way around this requirement, but it's a lot easier this way.

While you're at it, turn off or delete the mailer action adapter. It's harmless, but it would be a distraction.

That's it. We now have a form that matches our database table.

Inserting rows

In this step, we create a method to insert a row, and show how to make of it.

Now, inside the Management Interface, in your form folder, create a Z SQL Method with the id *testCreateRow*.

Set the parameters:

Connection ID This should be the database connection you set up to allow access to your test database.

Arguments On separate lines, specify “string1” and “string2”. (Leave off the quotes.)

Then, in the big text area, specify the code:

```
insert into simple_db values (
    0,
    <dtml-sqlvar string1 type=string>,
    <dtml-sqlvar string2 type=string>
)
```

Note: always use <dtml-sqlvar ...> to insert your variables. It protects you against SQL-injection attacks by SQL quoting the values.

Now for a little magic: Z SQL Methods can pick up their arguments from REQUEST.form, which is exactly where Zope is putting your form variables when you submit a form. That means that you can just go to the [overrides] pane of your Form Folder and set *here/testCreateRow* as your After Validation Script.

Your form will now store its input into your SQL table! Add a few rows to check it out. Reading a Row, Filling in the Fields

If we want to update records, we’re going to have to get rows from our SQL table and use the columns to populate our form fields.

The SQL

Now, use the Management Interface to create, inside your form folder, a Z SQL Method named *testReadRow*. Set up the following parameters:

Connection ID Choose your test database adapter.

Arguments Just “uid”

Then, add the SQL Code:

```
select * from simple_db where
    <dtml-sqltest uid type="int">
```

The <dtml-sqltest ...> operator is a safe way to use user input for an SQL “where” test. The default test is “=”.

The Script

Let’s wrap this method in a simple Python script that will selectively use it. Create a Python Script with the id *formSetup* and the Python:

```
request = container.REQUEST
form = request.form

if form.has_key('uid') and not form.has_key('form.submitted') :
    res = context.testReadRow().dictionaries()
    if len(res) == 1:
        row = res[0]
        for key in row.keys():
            form[key] = row[key]
```

Let's deconstruct this code.

The if test:

```
if form.has_key('uid') and not form.has_key('form.submitted')
```

will make sure that this code does nothing if the form has already been submitted (we don't want to overwrite values the user just input). It also won't do anything if we don't have a "uid" variable in the form dictionary. (form.submitted is a hidden input that's part of every PFG form.)

If we have a uid variable and we won't be overwriting user input, then we call our SQL read method:

```
res = context.testReadRow().dictionaries()
```

This will return the results of our SQL query in the form of a list of dictionaries. The dictionary entries will be in the form columnid:value.

Note that the uid value is being passed via the request variable, and doesn't need to be specified.

The rest of the code checks to make sure that we got one result, and throws all of its key:value pairs into the form dictionary – just where our form will expect them.

The form

Now, go to the [override] pane of your form folder, and specify `here/formSetup` for your Form Setup Script.

Calling The Form

Hopefully, you've got a few rows in your table.

Now, try calling your form with the URL:

```
http://localhost/testfolder/myform?uid=1
```

Everything up to the question mark (the query string marker) should be the URL of your form folder. The "?uid=1" specifies that we want to use the data from the row where the uid is "1".

How would you actually get your users to such a URL? Typically, you'd have some sort of drill-down search that offered them a list of links constructed in this fashion.

Creating a drill-down template is left as an exercise for the reader.

Updating or inserting as necessary

In this step, we'll create an update SQL method and show how to selectively update or insert data.

Using the Management Interface, create a Z SQL Method inside your form folder with the id `testUpdateRow`. For its parameters, set:

Connection ID Choose your test database connection.

Arguments Add "uid", "string1" and "string2" on separate lines, without quotes.

Then, specify the SQL code:

```
UPDATE simple_db
SET
    <dtml-sqltest string1 type="string">,
    <dtml-sqltest string2 type="string">
WHERE <dtml-sqltest uid type="int">
```

Notice the use `<dtml-sqltest ...>` for the SQL set id=value lines. This is a hack that uses `sqltest` where we could have instead written lines like `"string1=<dtml-sqlvar string1 type=string">`.

Now, we’ve got to solve a problem. How do we update our table under some circumstances, and insert new values under others?

Remember how we set “-1” as the default value of our hidden “uid” form field? If we’ve read a record, uid will have changed to match a real row. If it’s “-1”, that means that we started with a clean form rather than values read from a table row.

Let’s use that knowledge in a simple switchboard script with the id `doUpdateInsert`:

```
request = container.REQUEST
form = request.form

if int(form.get('uid', '-1')) >= 0:
    # we have a real uid, so update
    context.testUpdateRow()
else:
    context.testCreateRow()
```

Now, go to the [overrides] pane of your form folder and set `here/doUpdateInsert` as the *AfterValidationScript*.

Note: Believe it or not ... you’re done.

Time to go back and repeat the process with your own table. Don’t forget to add lots of sanity-checking code along the way.

A note on security

It takes extra steps to secure a database connection and SQL methods.

If this is the first time you’ve worked with a Zope database connection, there’s an important security point you may not have considered:

Warning: Zope Database Connections and Z SQL Methods are not part of the Plone workflow.

This means that you may not depend on the Plone content workflow to provide security for these connections and methods. You must use the Zope security mechanisms directly to control access.

This is also true of Python scripts and other Zope-level objects you might create via the Management Interface. But Zope provides a safety net of security for most of those. There is no such automatic safety net for external RDBMS access methods.

The easiest way to do this is to use the Management Interface to visit the top-most folder of your form and use the Security tab to customize security. Look in particular for the Use Database Methods permission, and make sure it is not extended to any user role that should not have a right to read or update your external database.

Using GnuPG encryption

Description

The Gnu Privacy Guard may be used to encrypt emails sent by PloneFormGen.

Warning: Encryption is serious business, and this how-to does not teach you about it or about the Gnu Privacy Guard. You should develop expertise with both of these before attempting to enable PFG encryption.

Using GPG encryption with PloneFormGen requires:

- 1) That gpg be installed on your system and available on the search path or in a common location (e.g., /usr/bin);
- 2) That gpg, when executed as a subprocess of Zope/Plone, be able to find a public keyring;
- 3) That gpg, when executed as a subprocess of Zope/Plone, have the rights to read the public keyring;
- 4) That you, as administrator, understand how gpg works, and be able to maintain the public keyring.

PloneFormGen tries to find the gpg binary when it's installed, when the product code is refreshed, and when you restart Zope. If it finds it you will see an "encryption" field set in the mailer adapter form. If you don't see the "encryption" fieldset, it means PloneFormGen didn't find a gpg binary. Fix this by adding the path to the gpg binary to the PATH environment variable when you start Zope.

gpg will typically look for a public keyring in the current user's home directory, .gnupg subdirectory. If it's not finding it, consider the possibility that the user id you're using to maintain your keys isn't the same one that you're using to run Zope. You may need to use the GNUPGHOME environment variable to point to your .gnupg directory. Make sure your Zope process can open the directory and read the file!

Note: Windows

gpg can work in a Windows environment, but you'll need to have a command-line gpg. <http://www.cygwin.com/> is a good, free source.

Embedding PloneFormGen forms

Description

PloneFormGen forms may be rendered from other templates, viewlets, and portlets.

Caveat: This feature should be considered beta quality. I've written code that takes advantage of it, and you shouldn't be afraid of it, but take care to test thoroughly. There may be certain types of contexts for rendering the form with implications that I haven't taken into consideration.

To insert the form into an arbitrary template, use the 'embedded' browser view:

```
<tal:block tal:replace="structure path/to/form/@@embedded"/>
```

If you are including the form on a page that features another form, you'll probably need to set a prefix on the 'embedded' view to disambiguate submissions:

```
<tal:block tal:define="form nocall:path/to/form/@@embedded;
    dummy python:form.setPrefix('mypfg') "
    tal:replace="structure form"/>
```

Or if you are using a view class, you could define a method like:

```
from Products.CMFCore.utils import getToolByName
def render_form(self):
    portal = getToolByName(self.context, 'portal_url').getPortalObject()
    form_view = portal.restrictedTraverse('path/to/form/@@embedded')
    form_view.prefix = 'mypfg'
    return form_view()
```

(Note that `restrictedTraverse` expects a path relative to the object you are calling it on, with no initial slash.) And then in the associated template:

```
<tal:block tal:replace="structure view/render_form"/>
```

By default the embedded form uses the current URL as the form’s ‘action’ parameter. When the form is rendered upon submission, it will perform validation, run the normal action adapters, and redirect to the success page as normal. If you want to submit to the form’s real location or somewhere else, you can override the action by setting the ‘action’ attribute on the ‘embedded’ view.

Known limitation: Embedded forms have no way of injecting JavaScript or CSS into the page head like their standalone counterparts.

Adding Custom Fields, Action Adapters or Thanks Pages

Description

You may add custom fields, action adapters and thanks pages to PloneFormGen. By far the easiest way to do this is to derive a subclass from one of the field types in `fieldsBase` or an action adapter from `actionAdapter.FormActionAdapter`.

When PFG is installed, or reinstalled, it will automatically add to its available fields, adapters and thanks pages list any installed Archetypes content type that implements one of:

- `Products.PloneFormGen.interfaces.actionAdapter.IPloneFormGenActionAdapter`
- `Products.PloneFormGen.interfaces.field.IPloneFormGenField`
- `Products.PloneFormGen.interfaces.thanksPage.IPloneFormGenThanksPage`

Also, the Archetypes class *must* specify a `meta_type` in the class definition that matches the `meta_type` defined in its GS type declaration. Otherwise, it won’t be found.

Creating content from PFG

Description

This how-to covers simple creation of portal content from PloneFormGen. We’ll create web pages from sample form submissions.

A question that's come up frequently on IRC and the Plone users' mailing list is "How do I create an event, news item, page, or some other content item from PloneFormGen? It's common that there's some security need or extra content needed that prevents just using Plone's "add item."

This is actually very easy if you know a little Python and are willing to learn something about the content items you want to create.

Please note that I'm not going to show you how to create new content types here. Just how to use PFG to create content objects from existing types. If you want to create new content types, learn to use Dexterity.

Your first step should be to determine the attributes you want to set in the new content item and how they'll map from your form fields.

In this case, we're going to use the sample contact form created when you first create a form folder to create a page (Document).

Our mapping of form fields to content attributes will look like this:

Form Field	Document Attribute
Your E-Mail Address (replyto)	Description
Subject (topic)	Title
Comments (comments)	Body text

Note that for each form field, we've determined its ID in the form. We'll use those to look up the field in the form submission.

Next, we need to learn the methods that are used to set our attributes on a Document object. How do you learn these? It's always nice to read the source, but when I'm working fast, I usually just use DocFinderTab and look for "set*" methods matching the attributes.

Now, determine where you want to put the new content. That's your target folder. It's convenient to locate that folder in a parent folder of the form object, as you may then use the magic of acquisition to find it without learning how to traverse the object database.

Now, in the form folder, we add a "Custom Script Adapter" - which is just a very convenient form of Python script. Then, just customize the script to look something like the following:

```
# Find our target folder from the context. The ID of
# our target folder is "submissions"
target = context.submissions

# The request object has a dictionary attribute named
# form that contains the submitted form content, keyed
# by field name
form = request.form

# We need to engineer a unique ID for the object we're
# going to create. If your form submit contained a field
# that was guaranteed unique, you could use that instead.
from DateTime import DateTime
uid = str(DateTime().millis())

# We use the "invokeFactory" method of the target folder
# to create a content object of type "Document" with our
# unique ID for an id and the form submission's topic
# field for a title.
target.invokeFactory("Document", id=uid, title=form['topic'])

# Find our new object in the target folder
obj = target[uid]
```

```
# Set its format, content and description
obj.setFormat('text/plain')
obj.setText(form['comments'])
obj.setDescription(form['replyto'])

# Force it to be reindexed with the new content
obj.reindexObject()
```

That's it. This will really work.

Security

At the moment, the person that submits your form will need to be logged in as a user that has the right to add pages to the target folder, then change their attributes. You may need to allow other users (even anonymous ones) to submit the form. That's where the Proxy role setting of the custom script adapter comes in. You may change this setting to Manager, and the script will run as if the user has the manager role - even if they're anonymous.

I hope it's obvious that you want to be very, very careful writing a script that will run with the Manager role. Review it, and review it again to make sure it will do only what you want. Never trust unchecked form input to determine target or content ids.

If I'm doing this trick with a form that will be exposed to the public, I often will use a Python script rather than the custom script adapter, as it allows me to determine the proxy role for the script more precisely than choosing between None and Manager. I may even create a new role with minimal privileges, and those only in the target folder. Credit!

Note: A big thank's to Mikko Ohtamaa for contributing the Custom Script Adapter to PloneFormGen.

Custom mailer script

Description

Customizing email output from PloneFormGen

Introduction

Below is an email script example to customize how PloneFormGen generates the email output.

Installation instructions

Go to form, on the contents tab remove the existing Mailer item.

Choose create new... Custom script adapter. Pick any name.

For the script, set Proxy role: Manager.

Fix the email addresses in the script below.

Paste the code to the script body field.

Save.

Test.

Example script

```
from Products.CMFCore.utils import getToolByName

mailhost = getToolByName(ploneformgen, 'MailHost')

subject = "Email subject"

# Use this logger to output debug info from this script if needed
import logging
logger = logging.getLogger("mailer-logger")

# Create a message body by appending all the fields after each another
# This includes non-functional fields like labels too
message=""
for field in ploneformgen.fgFields():
    label = field.widget.label.encode("utf-8")
    value = str(fields[field.getName()])

    # For non-functional fields draw a custom separator line
    if not field.widget.blurable:
        value = "-----"

    # Format lists on the same row
    try:
        if (value[0] == "["):
            value = value.replace("'", " ")[1:-1]
    except IndexError:
        # Skip formatting on error
        pass

    #remove last ':' from label
    if (label[-1] == ":"):
        label = label[0:-1]

    message += label + ": " + value + "\n\n"

source = "noreply@example.com"
receipt = "info@example.com"

mailhost.send(message, receipt, source, subject=subject, charset="utf-8", )
```

Fail-safe email sending

By default if SMTP server rejects the message sent by PloneFormGen the page will crash with an exception. Possible reasons for SMTP failure are

- SMTP server is down
- SMTP server is overloaded
- SMTP server spam protection is giving false positives for your email sending attempts

If you have a situation where gathering the data is critical, the following process is appropriate:

- Use save data adapter to save results
- Use a custom email sender script adapter to send email and even if this step fails then the data is saved and no exception is given to the user

Example PloneFormGen script adapter (using proxy role Manager):

```
# -*- coding: utf-8 -*-
from Products.CMFCore.utils import getToolByName

# This script will send email to several recipients
# each written down to its own email field
# whose id starts with "email-"
emails = []

for key in fields:
    if key.startswith('email-'):
        if fields[key] != '':
            emails.append(fields[key])

mailhost = getToolByName(ploneformgen, 'MailHost')

subject = "Huuhaa"

# Custom message with a name filled in
message = u"""Hello,

Thanks for participating %s !
Cheers,
http://www.opensourcehacker.com
"" % (fields['etunimi'])

source = "info@opensourcehacker.com"

for email in emails:
    try:
        mailhost.send(message, email, source, subject=subject, charset="utf-8", )
    except Exception:
        pass
```

Using TinyMCE as visual editor

A user manual for content creators

Introduction

Introduction to TinyMCE.

TinyMCE is a platform independent web based JavaScript HTML WYSIWYG editor. What this means is that it will let you create html content on your web site.

TinyMCE supports most modern operating systems and browsers. Some examples are: Mozilla, Internet Explorer, Firefox, Opera, Safari and Chrome. TinyMCE has a large userbase and an active development community.

TinyMCE is the default visual editor since Plone 4.0.

Note: Browsers on mobile devices were, until recently, limited in their support for editors like TinyMCE. The situation is improving all the time. If your mobile device supports installing different browsers, it can help to check if modern browsers like Chrome or Firefox give better results when working on a tablet or smartphone.

Updating to the latest version is usually a good idea.

Basics

Basic options of TinyMCE.

The default TinyMCE editor will look like this:

```
*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Open test browser
    Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
    Import library  Remote  ${PLONE_URL}/RobotRemote

    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Remote ZODB Setup  ${FIXTURE}

    ${language} =  Get environment variable  LANGUAGE  'en'
    Set default language  ${language}

    Enable autologin as  Manager
    ${user_id} =  Translate  user_id
    ...  default=jane-doe
    ${user_fullname} =  Translate  user_fullname
    ...  default=Jane Doe
    Create user  ${user_id}  Member  fullname=${user_fullname}
```

```

Set autologin username  ${user_id}

Test Teardown
Run keyword if  sys.argv[0].startswith('bin/robot')
...           Remote ZODB TearDown  ${FIXTURE}

Suite Teardown
Run keyword if  not sys.argv[0].startswith('bin/robot')
...           Teardown Plone Site
Run keyword if  sys.argv[0].startswith('bin/robot')
...           Close all browsers

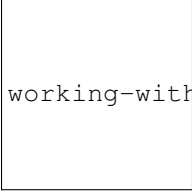
```

```

*** Test Cases ***

Show TinyMCE
Go to  ${PLONE_URL}
Click element  css=#contentview-edit a
Wait until element is visible
...  css=#mceu_16-body
Capture and crop page screenshot
...  ${CURDIR}/../../_robot/tinymce.png
...  css=#formfield-form-widgets-IRichText-text

```



working-with-content/using-tinymce-as-visual-editor/../../

On top you can see the toolbar, below the text area with the actual content you are editing and at the bottom a status bar. If you drag the lower right corner you can make the editor window bigger or smaller.

The editor allows you to style text, add images and links, add tables and more.

Note: Although it may look like a word-processor on your desktop computer, editing text for the web is slightly different. You would create headlines by selecting the “Formats” dropdown, and then choose “Header 1” (biggest) to “Header 6” (smallest).

You would **not** choose the font and size directly, those are set up by the theme. This will ensure a consistent look over your site, and it is also important to make content available to different devices (phones, tablets) and to ensure the best result for people with disabilities.

The toolbar icons and dropdowns generally work as you would expect.

Some of the more interesting ones are explained in the sections *Inserting images*, *Inserting links* and *Inserting tables*.

If you want to get at the pure HTML source code of your text, just use the “Tools” menu.

Inserting Images

```

*** Settings ***

Resource  plone/app/robotframework/server.robot

```

```
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Open test browser
    Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
    Import library  Remote  ${PLONE_URL}/RobotRemote

    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Remote ZODB SetUp  ${FIXTURE}

    ${language} =  Get environment variable  LANGUAGE  'en'
    Set default language  ${language}

    Enable autologin as  Manager
    ${user_id} =  Translate  user_id
    ...  default=jane-doe
    ${user_fullname} =  Translate  user_fullname
    ...  default=Jane Doe
    Create user  ${user_id}  Member  fullname=${user_fullname}
    Set autologin username  ${user_id}

Test Teardown
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Remote ZODB TearDown  ${FIXTURE}

Suite Teardown
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Teardown Plone Site
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Close all browsers
```

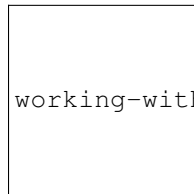
```
*** Test Cases ***

Show TinyMCE image
    Go to  ${PLONE_URL}
    Click element  css=#contentview-edit a
    Wait until element is visible
    ...  css=#mceu_16-body
```

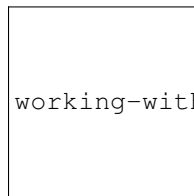
```
Capture and crop page screenshot
... ${CURDIR}/../../../../_robot/tinymce-imgbutton.png
... css=#mceu_15

Click element  css=#mceu_15 button
Capture and crop page screenshot
... ${CURDIR}/../../../../_robot/tinymce-imgdialog.png
... css=div.plone-modal-content
```

The TinyMCE editor allows you to insert image files stored in Plone into your document, using the Image button on the TinyMCE toolbar:



Clicking this button launches the Insert Image dialog:



There are three ways to select images:

Internal image This means the image is already on the site somewhere. You can search on it by title or description, or navigate to it.

Upload This means using an image file you already have on your computer. The image will be uploaded to the same folder as the content item you are editing.

External Image This means specifying the URL of an image that is elsewhere on the web.

For all three methods, you can set the Title, ALT text (this is important for non-sighted users, make this a description of the image) and the alignment. Aligning “inline” means the image will appear exactly where you put it, in the middle of a sentence if wanted. Aligning “left” or “right” will make the image go to the side of the paragraph, and text will flow around it.

For Internal and Uploaded images, you can also select the size. These sizes come from the range of sizes that you (or your site administrator) has set in the *control panel*. Plone automatically generates the different sizes when you upload an image.

Note: Be careful with the size “original”. Modern cameras and smartphones take high-resolution pictures, much higher than is usually needed in a website. Larger pictures take longer to download, making your site slower to appear.

Inserting Links

```
*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Open test browser
    Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
    Import library  Remote  ${PLONE_URL}/RobotRemote

    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Remote ZODB SetUp  ${FIXTURE}

    ${language} =  Get environment variable  LANGUAGE  'en'
    Set default language  ${language}

    Enable autologin as  Manager
    ${user_id} =  Translate  user_id
    ...  default=jane-doe
    ${user_fullname} =  Translate  user_fullname
    ...  default=Jane Doe
    Create user  ${user_id}  Member  fullname=${user_fullname}
    Set autologin username  ${user_id}

Test Teardown
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Remote ZODB TearDown  ${FIXTURE}

Suite Teardown
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Teardown Plone Site
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Close all browsers
```

```
*** Test Cases ***
```

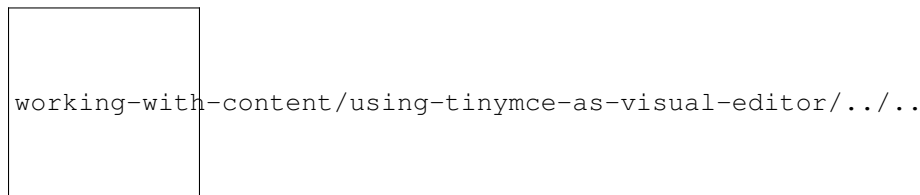
```

Show TinyMCE insert links
Go to  ${PLONE_URL}
Click element  css=#contentview-edit a
Wait until element is visible
...  css=#mceu_16-body
Capture and crop page screenshot
...  ${CURDIR}/../../_robot/tinymce-linkbutton.png
...  css=#mceu_14

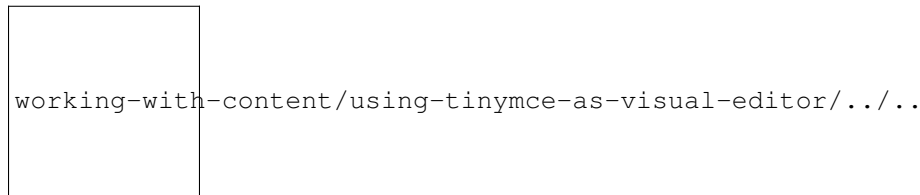
Click element  css=#mceu_14 button
Capture and crop page screenshot
...  ${CURDIR}/../../_robot/tinymce-linkdialog.png
...  css=div.plone-modal-content

```

Select a word or phrase and click on the *Insert/edit link* icon:



The Link dialog will appear:



Here, you can fill in the information on where you want to link to:

Internal Link

You can search for, or navigate to, the content item that you want to link to. Furthermore, you can set a “target”: open in the same browser window, or a new one. In general, it is considered good etiquette to always open internal links in the same window. Setting a descriptive Title for the link is helpful if the item you are linking to does not have a distinctive title of its own.

Upload

The Upload tab lets you upload a PDF or Office document or other file. It will be stored in the same Folder as the content item you are editing.

External Link

When linking to external sites, make sure you include the *complete* link, including the “http//” or “https://” part, otherwise it will be interpreted as a *relative* link within your own site. Again, you can set a Target and Title. Opinions differ on whether you should open an external link in a new window or not; ask if your organisation has a policy on this.

Email

This tab lets you create a `mailto:` link, which will open in the user's email program. You can optionally set a subject for the email, although your visitor will always be able to override it. setting the Email Subject is more a helpful suggestion.

Anchors

Anchors are like position markers within a document, based on headings, subheadings, or another style set within the document. You can also set Anchors at arbitrary positions in a document. Plone automatically creates Anchors for headings and subheadings, and thus you can directly link to a chapter in a long web document.

Inserting Tables

```
*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Open test browser
    Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
    Import library  Remote  ${PLONE_URL}/RobotRemote

    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Remote ZODB Setup  ${FIXTURE}

    ${language} =  Get environment variable  LANGUAGE  'en'
    Set default language  ${language}

    Enable autologin as  Manager
    ${user_id} =  Translate  user_id
    ...  default=jane-doe
    ${user_fullname} =  Translate  user_fullname
    ...  default=Jane Doe
```



```

Create user  ${user_id}  Member  fullname=${user_fullname}
Set autologin username  ${user_id}

Test Teardown
  Run keyword if  sys.argv[0].startswith('bin/robot')
  ...            Remote ZODB TearDown  ${FIXTURE}

Suite Teardown
  Run keyword if  not sys.argv[0].startswith('bin/robot')
  ...            Teardown Plone Site
  Run keyword if  sys.argv[0].startswith('bin/robot')
  ...            Close all browsers

```

```

*** Test Cases ***

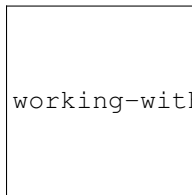
Show TinyMCE insert tables
  Go to  ${PLONE_URL}
  Click element  css=#contentview-edit a
  Wait until element is visible
  ...  css=#mceu_16-body

  Click element  css=#mceu_21 button
  Mouse over  css=#mceu_44
  Capture and crop page screenshot
  ...  ${CURDIR}/../_robot/tinymce-table.png
  ...  css=#content

```

Tables are handy for tabular data and lists.

To add a table, put your cursor where you want it and click the *Table* dropdown menu.



working-with-content/using-tinymce-as-visual-editor/../../

There are various options to choose a *style* for the table, insert rows and columns, and set properties on the individual cells. Plone comes with a few basic styles for tables, but you (or your site administrator) will most likely want to provide some extra CSS classes to make them look better.

Note: Creating and managing tables in HTML has historically been *awkward*. People tend to mis-use them for layout purposes, which you should not do.

Use tables **only** for tabular data. And, as rule of thumb, try to keep it to a very small number of rows and columns.

If you want to present information that is mostly tabular, such as larger amounts of statistical data, there are various add-ons to help you do that. These will generate nicer tables, and are easier to work with both for content editors and visitors to your site. Visitors will be able to sort tables and use a quick search to locate individual cells, for instance.

Collaboration and Workflow

Learn how to share and control access to your content by using the Sharing tab and the State menu.

Basic Publication States

The publication control system for Plone is very flexible, starting with basic settings for making an item private or public.

In the toolbar, for any content type –folders, images, pages, etc., and any specialized content types – there is an item for publication state. This *state* menu has settings for controlling publication state:

```
*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Open test browser
    Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
    Import library  Remote  ${PLONE_URL}/RobotRemote

    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Remote ZODB Setup  ${FIXTURE}

    ${language} =  Get environment variable  LANGUAGE  'en'
    Set default language  ${language}

    Enable autologin as  Manager
    ${user_id} =  Translate  user_id
    ...  default=jane-doe
    ${user_fullname} =  Translate  user_fullname
    ...  default=Jane Doe
    Create user  ${user_id}  Member  fullname=${user_fullname}
    Set autologin username  ${user_id}

Test Teardown
```

```

Run keyword if sys.argv[0].startswith('bin/robot')
...           Remote ZODB TearDown  ${FIXTURE}

Suite Teardown
Run keyword if not sys.argv[0].startswith('bin/robot')
...           Teardown Plone Site
Run keyword if sys.argv[0].startswith('bin/robot')
...           Close all browsers

*** Test Cases ***

Create sample content
Go to  ${PLONE_URL}

    ${item} = Create content  type=Document
    ... id=samplepage  title=Sample Page
    ... description=The long wait is now over
    ... text=<p>Our new site is built with Plone.</p>

Show state menu
Go to  ${PLONE_URL}/samplepage

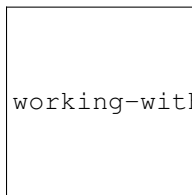
Wait until element is visible
...  css=span.icon-plone-contentmenu-workflow
Click element  css=span.icon-plone-contentmenu-workflow

Wait until element is visible
...  css=#plone-contentmenu-workflow li.plone-toolbar-submenu-header

Mouse over  workflow-transition-publish
Update element style  portal-footer  display  none

Capture and crop page screenshot
...  ${CURDIR}/../../../../_robot/workflow-basic.png
    ...  css=#content-header
    ...  css=div.plone-toolbar-container

```



working-with-content/collaboration-and-workflow/../../../../ro

The toolbar will show the current publication state for the content item, such as *State: Published*, as shown above. Private is the initial state when you create a content item – a page, a news item – and in the private state, as the name indicates, the content item will generally not be available to visitors to the website.

The *Publish* menu choice will make the content item available on the web site to anonymous visitors. The *Submit for publication* menu choice is used on web sites where there are content editors who must approve items for publication, as discussed below.

Also, and this will be very important, certain content types, such as news items and events, will not appear on the website as you expect, until they are explicitly *published*.

Note: Store this in your memory: **Publication state is important!**

Publication state can be changed only by users whose accounts have the necessary permissions. The menu choices in the state menu will reflect existing permissions settings. For example, in a big newspaper web site, a reporter could add pages for news articles, but the state menu will not show a *Publish* menu choice, only a *Submit for publication* menu choice. This is because a reporter must submit articles up the line to the editorial staff for approval before publication. If your account has the permissions, however, the *Publish* menu choice will appear and you can simply publish in one step.

For an editor, a content item that has been submitted may be *published* or *rejected*, either outright, because it is an inappropriate submission for the situation, or for the more typical reason that the content item needs revision.

After a content item has been *published*, it may be *retracted*, to change the state back to *public draft* state, or *sent back* to private, if desired. The menu choices in the state menu will change accordingly:

```
Show sendback
Go to  ${PLONE_URL}/samplepage

Wait until element is visible
...  css=span.icon-plone-contentmenu-workflow
Click element  css=span.icon-plone-contentmenu-workflow

Wait until element is visible
...  css=#plone-contentmenu-workflow li.plone-toolbar-submenu-header

click link  workflow-transition-submit


Go to  ${PLONE_URL}/samplepage

Wait until element is visible
...  css=span.icon-plone-contentmenu-workflow
Click element  css=span.icon-plone-contentmenu-workflow

Wait until element is visible
...  css=#plone-contentmenu-workflow li.plone-toolbar-submenu-header

Mouse over  workflow-transition-reject
Update element style  portal-footer  display  none

Capture and crop page screenshot
...  ${CURDIR}/../../../../_robot/workflow-reject.png
...  css=#content-header
...  css=div.plone-toolbar-container
```



working-with-content/collaboration-and-workflow/../../../../_ro

Instead of completely deleting items in your site that have become obsolete or undesired for some reason, you can also think about retracting (“unpublishing”), or setting to *private*, any content .

Setting to *private* will take the item from public view and from showing up in search results, but will keep it around in case the format or the actual material (text, images, etc.) is needed later, or you later change your mind and want to

re-publish the content.

Note: Content that was published once on a public website may have been indexed by search engines. Unpublishing will make it invisible to direct visitors to *your* site, but search engines often keep a copy of it in their indexes. Then again, the same is valid for *deleting* content.

The decision to delete or to set to *private* may depend on whether or not the content exists elsewhere, on another computer or in your company central data storage. Having large amounts of *private* content on a site might confuse editors, and it will take up some disk space on your webserver.

Advanced Control

```
*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  1200
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Open test browser
    Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
    Import library  Remote  ${PLONE_URL}/RobotRemote

    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Remote ZODB Setup  ${FIXTURE}

    ${language} =  Get environment variable  LANGUAGE  'en'
    Set default language  ${language}

    Enable autologin as  Manager
    ${user_id} =  Translate  user_id
    ...  default=jane-doe
    ${user_fullname} =  Translate  user_fullname
    ...  default=Jane Doe
    Create user  ${user_id}  Member  fullname=${user_fullname}
    Set autologin username  ${user_id}
```

```
Test Teardown
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...            Remote ZODB TearDown  ${FIXTURE}

Suite Teardown
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...            Teardown Plone Site
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...            Close all browsers
```

The publication control system, under the advanced menu, has sophisticated features for setting availability by date and by context.

The *state* Toolbar menu has an *advanced...* choice:

```
*** Test Cases ***

Create sample content
    Go to  ${PLONE_URL}

    ${item} = Create content  type=Folder
    ... id=documentation  title=Documentation
    ... description=Here you can find the documentation on our new product

Show state menu
    Go to  ${PLONE_URL}/documentation

    Click link  css=#plone-contentmenu-workflow a

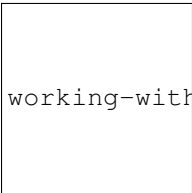
    Wait until element is visible
    ...  css=#plone-contentmenu-workflow li.plone-toolbar-submenu-header

    Mouse over  workflow-transition-advanced
    Update element style  portal-footer  display  none

    Capture and crop page screenshot
    ...  ${CURDIR}/../../../../_robot/workflow-advanced-menu.png
    ...  css=#content-header
    ...  css=div.plone-toolbar-container

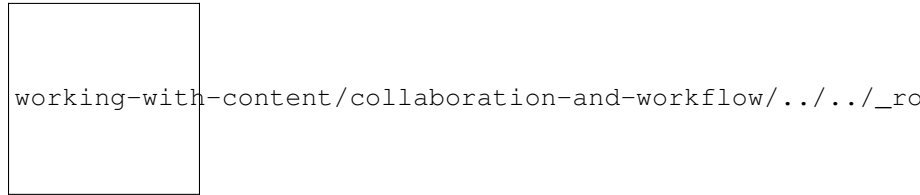
    Click link  workflow-transition-advanced
    Wait until element is visible
    ...  css=div.plone-modal-content

    Capture and crop page screenshot
    ...  ${CURDIR}/../../../../_robot/workflow-advanced.png
    ...  css=div.plone-modal-wrapper
```



working-with-content/collaboration-and-workflow/../../../../_ro

which brings up the *advanced* state panel:



Below an explanation section at the beginning of the panel, there is a check box showing the content that will be affected by this change of publication state. Here it shows we are working on the “Documentation” folder.

Note: A brief aside: you may have noticed that, apart from the checkbox to the left, there is also a checkbox above, next to the headers of the *Affected content* section. The reason is that this same *advanced* state panel can be used to make the changes described below to a whole list of unrelated content items simultaneously, in bulk, from the *contents* tab of a folder. More on this later.

The next field, *Include contained items*, is a check box for controlling whether the state change affects this item only (the “Documentation” folder) or the items it contains and all of any subfolders and other contained items.

This is an important check box.

It lets you change the availability of an entire section of a website. For example, the “Documentation” folder could contain four subfolders, with images, files, and other content that has been kept *private* during the initial work to build up this content. All of it could be immediately made public – it could be *published* – by checking this box and checking *Publish* at the bottom before saving.

Likewise, the *Submit for publication* choice would be used on a web site where editors control ultimate publication.

And you can do the reverse, of course: make an entire section *private*. For example, if an automobile rental agency decided to remove a car model from its fleet, an entire section of their website devoted to this car model, with several subfolders full of pages, images, and files, could be set to *private*.

The next two date fields are for *Publishing date* and *Expiration date*. Their meanings are straightforward. If there is a window of time, for which a content item or a set of content items is valid for publication, it may be set with these fields.

A *comment* lets you attach an explanation to all content affected by the state change.

This is especially useful when several people are working on a website, and a person less familiar with an area of the web site looks at content and wonders why it isn’t published. They wonder, “This information looks good. Why isn’t it published already?” Then they read a comment that says something like, “Don’t publish until Richard checks on copyright issues regarding the items described here.” Using comments is a good idea for sensitive information, even if you are the only person working on the web site, because you might forget why you made a decision about publication state.

Finally, at the bottom there is a choice of several available states for this action. It will vary, depending on the present state of the item. For example, if the item is currently in a published state, there won’t be a choice for *publish*, if the item is presently in a *private* state, there won’t be a choice for *make private*, etc. If an item is published already, there will be choices in this bottom part of the panel for *reject* and *retract*, for “unpublishing” at item, setting it back to *public draft* or then to *private* state.

Workflow Policies

Workflow policies allow a site administrator to create a formalized system for controlling publication and content management as a step-by-step flow involving different users in set roles.

Workflow is a relatively advanced subject.

It involves creation of a more regimented control of content creation, review, and publication. If you have a user account on a typical small Plone site, you will probably not encounter custom workflow policies, because there isn't a need for this more sophisticated control.

But, the potential is there for using this functionality, as it is built in to Plone, and larger organisations often use it to model their internal structure and business logic.

For an introduction to the workflow concept, consider an example involving a web site for a newspaper business, for which these different groups of people are at work:

Reporters Can create stories, but can only submit them for review.

Editors Can review stories, but can't publish completely. They send positively reviewed and edited stories up the line for further approval.

Copy Editors Do final fact checking, fixes, and review, and may publish stories.

A *workflow policy*, sometimes abbreviated to *workflow*, describes the constraints on state-changing actions for different groups of people. Once the workflow policy has been created, it needs to be applied to an area of the website for the rules to take effect.

In the example of the newspaper web site, a workflow policy would be set up and then applied to the folders where reporters do the work of adding news articles.

Then, reporters would create stories and send them up the line for review and approval:



Reporters would add news articles and would *submit* them (the *publish* menu choice is not available to them). Likewise, editors may *reject* the article for revision or they may, in turn, *submit* the article up the line to a copy editor for final proofreading and publication. In this newspaper business example, this policy could be called something like “Editorial Review Policy.”

Configuring a workflow policy is a matter of defining the scope of the workflow.

This is a web site administrator task.

There are two main ways to set workflow policies: by content type or by area of the site. Our newspaper may set their “Editorial Review Policy” on the content type “Article”, whereas they may have another workflow for Images since they go through a different process of review and approval.

But it is also possible to have different workflows for different sections of the website. The “Letters to the Editor” section may have a different workflow.

The web site administrator would use control panels of Plone to specify where on the web site the “Editorial Review Policy” applies, site-wide or to a subsection.

Plone comes with several useful workflow policies – the default one is a simple web publishing policy. Your web site administrator might employ a more specific policy, such as a policy for a community-based web site or a company

Intranet (internal web system). If so, you may need to learn some procedural steps to publishing, but these are just elaborations of principles in the default, basic workflow policy.

Collaboration through Sharing

```

*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Open test browser
    Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
    Import library  Remote  ${PLONE_URL}/RobotRemote

    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Remote ZODB SetUp  ${FIXTURE}

    ${language} =  Get environment variable  LANGUAGE  'en'
    Set default language  ${language}

    Enable autologin as  Manager
    ${user_id} =  Translate  user_id
    ...  default=jane-doe
    ${user_fullname} =  Translate  user_fullname
    ...  default=Jane Doe
    Create user  ${user_id}  Member  fullname=${user_fullname}
    Set autologin username  ${user_id}

Test Teardown
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Remote ZODB TearDown  ${FIXTURE}

Suite Teardown
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Teardown Plone Site
    Run keyword if  sys.argv[0].startswith('bin/robot')

```

... Close all browsers

The Sharing item on the Toolbar empowers you to collaborate with other users through the use of several built-in roles. Here, Jane Doe has created a Folder called “Documentation”, and has clicked on the “Sharing” option.

```
*** Test Cases ***

Create sample content
  Go to  ${PLONE_URL}

  ${item} = Create content  type=Folder
  ... id=documentation  title=Documentation
  ... description=Here you can find the documentation on our new product

Show sharing menu


  Go to  ${PLONE_URL}/documentation

  Click link  css=#contentview-local_roles a

  Wait until element is visible
  ...  css=#user-group-sharing-container

  Update element style  portal-footer  display  none

Capture and crop page screenshot
  ...  ${CURDIR}/../../../../_robot/sharing-menu.png
  ...  css=#content-header
  ...  css=div.plone-toolbar-container
```



working-with-content/collaboration-and-workflow/../../../../_ro

The default workflow for this Plone site has not been modified, so the folder is still “Private”. Only she can see the contents.

Now she wants to let her colleagues add content to the Documentation folder.

Taking a closer look at the available permissions, they are:

- **Can add** - When this permission is granted to a particular user (or group of users), that user can then add new content items. And since that user was also the creator of that content item, they will be able to edit it as they like.
- **Can edit** - When this permission is granted on a folder, the user can not only edit the Folder (its title and description) but can also edit any of the items in the folder. Note, however, the user is not allowed to delete any of the content. When this permission is granted on a Page, for example, the user can only edit that Page and none of the other items in the folder.
- **Can view** - When this permission is used on a folder or other item, the user can view the content but not make any changes.
- **Can review** - When this permission is granted, the user can publish items.

Note: Note: these permissions will override the default workflow permissions! For example, if you grant a user “Can view” permission on a Page that is in the Private state, that user will be able to see that Page.

Users and groups

Now, while it is handy to give one person the right permissions, it quickly becomes messy in a larger site.

Plone has a well-thought out system of *User and Group management* that can help. For instance, if you put the people that work on the Sports section of your news site into the group “Sport editors”, you can give them all the permissions they need at once.

And even more important, if Sally gets transferred from the Sports section to the Science section, you only have to remove her from one group, and add to another group, and all permissions on the whole site will work for her.

One final note: if the “Documentation” folder was not in the published state, and you had also not given your colleague the ‘view’ permission on the folder, but you had given that permission on a specific Page in this folder, he or she would have to know the exact URL to the Page to see it.

Permissions are very specific in Plone! In practice, it is usually clearer to give permission on a folder-by-folder base and not per document, as this becomes hard to maintain. Then again, if you need the permissions and the security to be this fine-grained, you can!

How a folder’s workflow state affects its content

Description

On this page we are referring to workflow states and their effects on content as they are configured in a default Plone installation. Your specific site may have custom workflows, and the following discussion may or may not apply.

When it comes to the *Private* state, **Folders** are somewhat special. Changing a folder to (or leaving it in) *private* state has the following effects:

- The folder *as well as all its contents* are taken out of the navigation and site map for anonymous users, and also for logged-in users who don’t have permission to see private content. This means that all these users will not be able to find either the folder or any of its contents through any of the navigation menus. Of course, this includes external search engine robots.
- The folder itself can not be viewed by anonymous users, or by logged-in users who do not have permission to see private content. This is true even if an anonymous user, for example, had the direct URL to the folder, which would be the case if a link to the folder was part of the content body of a page in a different section of the same site or even a different site. Clicking such a link would result in being redirected to the login form.
- However, any **published** content of a **private** folder (or even of any of its sub-folders) **will** appear in the site search, even for anonymous users.
- Also, anonymous users who know the URL of a *published* content item inside a *private* folder will be able to view this content. Consequently, if a link to any such published content of a private folder is embedded in any part of the same site or external site, that content will be viewable by anyone.

Thus, putting a folder in the private state is not a guarantee of security for any of its contents. Unless, of course, all the content has been made private, as well. This can be done in bulk and in a single step, as described in *Advanced Control*.

This is especially true of a folder's default item view (see *Setting an Individual Content Item as the View for a Folder*). If the contained item that is set as the folder's default view is published, then the folder will in a sense be public as well, even if its own state has been set to private. However, the folder will still be hidden from navigation for anonymous users.

When it comes to the folder default item view, care must be taken to have clarity on whether the desired workflow state is set on the folder, the default view item, or both.

A caveat: Images and Files

When discussing **published** content of a **private** folder above, we glossed over an important assumption: namely, that all content items actually have a **published** state. This assumption is actually incorrect. The *Image* content type and the *File* content type do not have the *State* menu (in a default Plone installation). Thus, they can not be made public or private or any other state. Instead, Images and Files *inherit* their state from the container in which they find themselves. Therefore an image in a private folder will be private; an image in a public folder will be public.

It is possible to bypass this inheritance of a folder's workflow state by contained images and files.

One of the workflows shipped with Plone by default is called "Single State Workflow". To change the workflow for all Image content items, go to Content Settings on the Site Setup page. Select *Image* (or *File*) in the top dropdown menu, and then "Single State Workflow" from the *New workflow* dropdown menu. Once you click *Apply changes*, all Image content items will acquire the new workflow, and in particular, they will all be in published state, and will not inherit the containing folder's workflow state.

Some site administrators prefer all **Images** to be published, but do have special workflow states for **Files**, as some files may only be accessible for logged-in users with a certain role. That is entirely possible using the above method, and setting an appropriate workflow on the type **File**

Using Listings & Queries (Collections)

Collections take advantage of the intelligence of Plone.

Think of them as automatically updated queries, with criteria that you define. As new pieces of content are added, they will show up in these Collections if they match the criteria.

Collections have gone through various iterations, since Plone 4.2 the so-called 'newstyle' collections are enabled by default. These are easy to create and maintain.

Introduction to (new-style) Collections

A Collection in Plone works much like a report or query does in a database. Use Collections to dynamically sort and display your content.

A **Collection** in Plone works much like a report or query does in a database. The idea is that you use a Collection to search your website based on a set of **Criteria** such as: content type (page, news item, image), the date it was published, or keywords contained in the title, description, or body.

Let's say you have a large catalog of photos and maps on your website. You can display them all at once by creating a link to the folder they're stored in. You could even create different links for subfolders if you've organized things that way. However, if your images and maps were spread out over the site in many folders this would quickly become cumbersome. Also, there is no way with normal folders to display different content, from different parts of your site based on things like:

- keywords in the title
- date of creation

- author
- type of content

The need for showing content in a variety of dynamic ways has given rise to **Collections** (formerly known as **Smart Folders**, or **Rich Topic** in older versions of Plone). Collections do not actually contain any content items themselves in the same way that a folder does. Instead it is the **Criteria** that you establish which determines what content appears on each Collection page.

Common applications for Collections are:

- News Archives
- Event Archives
- Photos Displayed by Date Range
- Content Displayed by Keyword

Creating a collection

When you add a new collection, you can select from a lot of different options.

The content type (Page, Event, Folder, etcetera), various dates, the location in the site and also keywords or tags are available as selection criteria.

While you're adding criteria, the number of content items that fit those criteria is dynamically updated.

By combining more criteria, you can create sophisticated queries, which will be automatically updated.

The combination shown below will always show upcoming events:

- content type is event
- the start date must be within 31 days from now

The resulting "Collection" works much like a Folder, only it is always up-to-date according to the criteria you set.

Please experiment with Collections, they are one of the most powerful features of Plone!

Before that, there were 'oldstyle' Collections. Setting them up was a bit less streamlined, but for power users they can have more flexibility, which is why you can enable them in your site if wanted.

Introduction to (old-style) Collections

A Collection in Plone works much like a report or query does in a database. Use Collections to dynamically sort and display your content.

A **Collection** in Plone works much like a report or query does in a database. The idea is that you use a Collection to search your website based on a set of **Criteria** such as: content type (page, news item, image), the date it was published, or keywords contained in the title, description, or body.

Let's say you have a large catalog of photos and maps on your website. You can display them all at once by creating a link to the folder they're stored in. You could even create different links for subfolders if you've organized things that way. However, if your images and maps were spread out over the site in many folders this would quickly become cumbersome. Also, there is no way with normal folders to display different content, from different parts of your site based on things like:

- keywords in the title
- date of creation
- author

Add Collection

Collection of searchable information

Default ■

Settings

Categorization

Dates

Creators

Title ■

my collection

Summary

Used in item listings and search results.

Search terms

Define the search terms for the items you want to list by choosing 1

Select... ▼

Select...

Text

- Description
- Searchable text
- Tag
- Title

Dates

- Creation date
- Effective date
- Event end date
- Event start date
- Expiration date
- Modification date

Metadata

- Creator
- Location
- Review state
- Short name (id)
- Type

☐ **Reversed order**

terms.

Search terms

Define the search terms for the items you want to list by choosing what to match on. The list of results will be dynam

Type ▼	Is ▼	Page ▼	Remove line
Select...		<input type="checkbox"/> Collection <input type="checkbox"/> Comment <input checked="" type="checkbox"/> Page <input type="checkbox"/> Event <input type="checkbox"/> File <input type="checkbox"/> Folder <input type="checkbox"/> Image <input type="checkbox"/> Link <input type="checkbox"/> News Item <input type="checkbox"/> Collection (old-style)	

Sort on Sortable Title ▼ ☐ Reversed order

Preview

Preview of at most 10 items.

1 items matching your search terms.

[Welcome to Plone](#) — last modified Apr 01, 2014
 Congratulations! You have successfully installed Plone. [/P](#)

**Body Text**

Search terms

Define the search terms for the items you want to list by choosing what to match on. The list of results will be dynamically updated.

Type ▼	Is ▼	Event ▼	Remove line
Event start date ▼	Within next ▼	31 days	Remove line
Select...			

Sort on Event start date ▼ ☒ Reversed order

Preview

Preview of at most 10 items.

- type of content

The need for showing content in a variety of dynamic ways has given rise to **Collections** (formerly known as **Smart Folders**, or **Rich Topic** in older versions of Plone). Collections do not actually contain any content items themselves in the same way that a folder does. Instead it is the **Criteria** that you establish which determines what content appears on each Collection page.

Common applications for Collections are:

- News Archives
- Event Archives
- Photos Displayed by Date Range
- Content Displayed by Keyword

Adding Collections

Collections (formerly called Smart Folders) are virtual containers of lists of items found by doing a specialized search.

Learning to think about content being stored wherever it is stored, and about using custom collections to gather up different “views” of the content, is an important step to using Plone most effectively. It is an intelligent system.

To add a collection, use the *Add new...* menu, as for adding other content types:

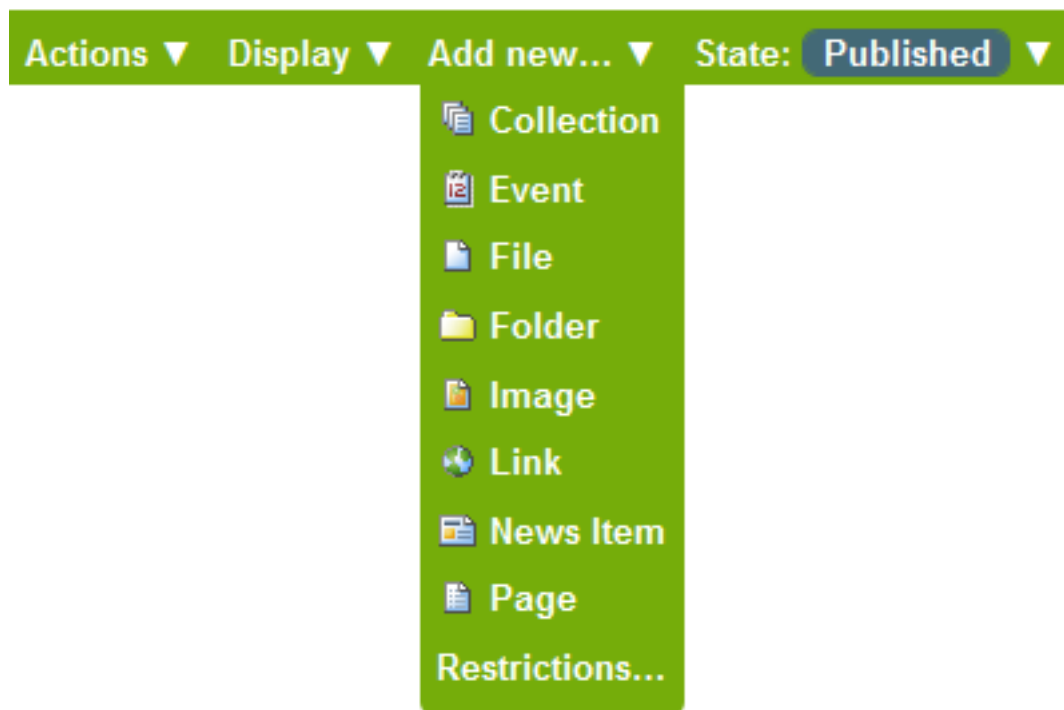


Fig. 3.2: p4_addnewmenu

You will see the Add*Collection*panel:

Below the title and description fields is a set of fields for specifying the format of search results returned by the search criterion for the new collection. The four fields in the panel above are in pairs. The top two fields let you limit the

Add Collection

An automatically updated stored search that can be used to display items matching criteria you specify.

Default

Categorization

Dates

Ownership

Settings

Title ■ (Required)
Title is required, please correct.

Description
Used in item listings and search results.

Body Text

Style...

B I [List Icons] [Link Icon] [Image Icon] [Table Icon] [HTML Icon]

☐ Limit Search Results
If selected, only the 'Number of Items' indicated below will be displayed.

Number of Items
0

☐ Display as Table
Columns in the table are controlled by 'Table Columns' below.

Table Columns
Select which fields to display when 'Display as Table' is checked.

Creation Date
Creator
Description
Effective Date
End Date
Expiration Date

>>
<<

Title

Save

Cancel

Fig. 3.3: p4_addcollection2

search results to a number of items that will be displayed. The bottom two fields let you control the order of the search result items.

Setting the search criterion

After setting the display configuration in the edit panel shown above, click the criteria tab to show the panel for setting search criteria:

Contents View Edit **Criteria** Sharing

Criteria for News and Events

No criteria defined yet. The search will not show any results. Please add criteria below.

— Add New Search Criteria —

Field name
The time and date an item was created

Creation Date ▼

- Creation Date
- Creator
- Description
- Effective Date
- End Date
- Expiration Date
- Item Type
- Location
- Modification Date
- Related To
- Search Text
- Short Name
- Start Date
- State
- Tags
- Title
- Title

☐

Save

Fig. 3.4: p4_collectionssearchcrit1 2

The top area of the panel, *Add New Search Criteria*, lets you set a field and a matching criterion. The bottom area, *Set Sort Order*, is a simple selection for a field to sort on:

The criteria types for matching data in content items depend on which field is selected.

After saving the collection, the search criteria will be applied and the results shown when the collection is clicked. You can create any number of collections for such custom views. For the butterfly example described above, in addition to a date constraint to find recent items, the categories field could be used to match color to have a series of collections for “Blue Butterflies,” “White Butterflies,” etc.

Multiple criteria can be used for a collection. For example, a collection called “Butterflies Photographed in the Last Month,” could be made by setting a criterion on Creation Date and on Item Type as Image. Such date-based collections are really effective to show up-to-date views of content that don’t require any administrative hand-work – once such a collection has been created, it will automatically stay up to date.

[Contents](#) [View](#) [Edit](#) [Criteria](#) [Sharing](#)

Criteria for News and Events

No criteria defined yet. The search will not show any results. Please add criteria below.

—Add New Search Criteria—

Field name
The time and date an item was created

Criteria type
Criteria does match

—Set Sort Order—

Field name
List Available Fields

Reverse
Reverse display order
☐

Fig. 3.5: p4_collectionssearchcrit2 2

Note: A collection doesn't behave like a normal folder, you can't add content items via the add item menu, as you can for a normal folder.

Adjusting the Display Settings

Learn how display settings can change the look of your Collection page

While the main power of Collections lies in Criteria, the display settings can make a big difference in the way your Collection will appear. All three of the settings we will cover in this section can be found by clicking the **Edit tab** of a Collection.

Inherit Criteria

By selecting the **Inherit Criteria** option, the Collection will inherit the Criteria from a parent Collection. This is only useful when using Sub Collections. If this is checked, you can create another Collection that is more specific than the Parent while still retaining the basic criteria of the Parent. A simple example might be a Parent Collection for displaying all Events in a site, and a Sub Collection that also displays Events (by inheriting Criteria) *but only* those Events with a particular keyword.

Limit Search Results

We can use Limit Search Results to limit the number of results that are Collection will display *per page*. This way if we have a Collection that is displaying News Items, we can limit the results to five or ten, instead of having it show all News Items on a single, large list.

Display as a Table

Display as a Table is simply another way to display the results of a Collection. Instead of having the Collection spit out the results in a list form, we can have it **generate a table** with the results, and set exactly what information about the results we want displayed. We customize the table by selecting the **Table Columns** from the left and clicking the right arrow button to move it over to the right. In the example above we chose to include the Title of the object, its Creator, and the Effective date. You can use any number of the columns, or all of them if you so choose.

When considering what to select, keep in mind that not all objects will have the information for every column available. For example, the **Start Date** and **End Date** only apply to Events. Therefore if you added these columns and your table included Pages as well as Events then the rows for the Pages would not have the Start and End Dates filled in. The other thing to consider is that the more columns you have showing the more crowded the table will become. The best rule of thumb is to only display what you absolutely need to display.

A few more notes on selecting columns: You can select more than one at a time by holding down the control key (Ctrl) while you click. If you want to remove a column, select it on the right and click on the left arrow button. Also you can add and remove columns by double clicking on their name.

Definition of Criteria

Definitions and examples of the different criteria fields available

The power of Collections most certainly lies with the Criteria fields. Mastering how to use the different Criteria will allow you to use Collections in several useful ways. In this section, we will use examples to illustrate the many ways of using Criteria.

Categories

The Category criterion allows you to search the **Category field** of objects. For this to work you must specify Categories for the content objects ahead of time (this is done through the Categorization tab on content objects). An example where you could use this is you want to create a Collection that would display all objects relating to the Category

Organization. As you can see in the image below, we are able to select the value *Organization* for our criterion. Then, by saving this criterion and viewing our Collection, the results would be all content objects we had designated with the Category *Organization*.

Once again the values available to you are completely dependent on what you have specified on your objects in the Categorization tab.

Creator

When using the Creator criterion, we are **filtering objects based on who created them**. This might be useful if you want to do a featured author section, where you would only want to display content on your site that has been created by a certain author.

As you can see we have several options for our criterion type. They allow us to restrict the creator to the person currently logged in, enter the name of another user as text, or to select users from a list.

If you want to display results from multiple users, you would need to use the **List of Values** option. Otherwise you would normally use the Text option unless the creator you wanted to select was yourself in which case you would use Restrict to Current User.

Description

The Description field is essentially a **search box type** criterion. However, instead of searching the title and body of a page, it will **only search for the text in the Description field** of a content object. This criterion is only really useful if you fill out the Description field consistently for all your content objects.

Location

Using the Location criterion is much like specifying a location when you search for a document on your hard drive. By specifying a Location criterion, **the results that are displayed in your Collection will only come from that location**, most commonly a Folder. This can be useful if you only want to display content that is in the About Us section of your site, for example. This is also useful for narrowing Collection results when combined with other criteria.

To specify a Location, simply click the **Add button**, which will pop up a new window showing you a directory of your site. If we follow our example and want to search the About Us section of our site, we would click the Insert button next to the About Us folder.

You can open folders to view content contained within them either by clicking the Browse button or directly on the title of the folder you want to open. You may also use the Search box to search for the Title of an object.

Search Text

The Search Text is a very useful criterion. It is similar to the search box on your site or an Internet search engine. It takes the text you specify and searches the Title, Description, and Body of all objects and returns **any that have the word or phrase you specify**. This is useful when you want to find objects that have to deal with a certain thing, especially if the word or phrase appears across many content types. Using training.plone.org as an example, if I want to create a Collection that displays all objects that reference the word Collections, I would use the Search Text criterion and specify *collections*. All Tutorials, Videos, Glossary items, etc with *collections* in the Title, Description, or Body would then appear in the Collection results.

Related To

The Related To field is another field, like Category, that **must be specified on a content object prior to being used for a Collection**. The Related To field on an object lets you specify which other objects in your site are similar or are relevant to the object you created. By specifying this field, when you create an object you can create a web of related content that will reference each other (think of a “see also” kind of function). When you have done this, you can use the Related To criterion in a Collection to display anything related to a specific object.

In this case we have specified that there are pages related to Our Staff, History, and the About Us Homepage. By selecting one or multiple values from this list, our Collection will display the pages that related to that Value.

If we selected History as the value we wanted, our Collection would show us everything that is related to the History page.

Keep in mind that the Related To Values list does not work based on which objects are related to content but on which objects have another object related **to it**. The Collection will display the results that are related to that value.

State

Using the State criterion is very simple. It allows us to **sort by published or private** state. It is a very good idea to restrict publicly viewable Collections **to filter on published**, so that no private content appears in the Collection results. Filtering on the Private state can be useful as well. For example, a site administrator might want to quickly see private content, so that they could determine what work needs to be done and what could be deleted.

Dates

You may have noticed that there are **several different dates available** to use as Criteria. Since there are such a large number of dates, they will be covered in *their own section of the manual*

Setting the Sort Order

Learn how to use the Sort Order feature to customize the order in which your results display

The Sort Order **determines the order the results of the Collection will be displayed in**. Sort Order allows you to sort by three main categories: text, object properties, and dates. When you sort by text, objects will be sorted in alphabetical order. When sorting by one of the object properties, we effectively are grouping objects together by the specified properties. When we sort by a date the results will be displayed with the most recent first (although there are many ‘dates’ in Plone). All Sort Orders are in Ascending Order unless the Reverse Order check box is selected. By checking this we can display in reverse order, or newest dates first, etc.

Dates

There are numerous Date options which will be explained in the next section of the manual.

Object Properties

Item Type

When sorting by Item Type, we end up with a Collection that has results that are grouped by Item Type. We would want to use this if we have a Collection that will return many different Item Types. This way we can make the Collection very easy to browse for the site visitor.

State

Sorting by State will display results grouped by the publishing state. Since there are only two States in the default configuration of Plone, there will only be Published and Private items. We can use this to separate all pages on our site and see what we have that is public (Published) and what we are hiding from the public eye (Private).

Category

Category Sort Order is useful when we want to display the objects on our site in a manner where they are grouped by the Category we placed them in. Keep in mind, for sorting by Category to even be remotely useful, you must have specified the Category on several objects. If you have not specified any Categories, then sorting by Categories will do nothing.

Related To

The Related To Sort Order will actually apply a criterion to your Collection. It limits to the results to only those that have Related To information specified on their properties.

Text

Short Name

Sorting by the Short Name is the same as putting the result objects in alphabetical order. By default Plone sets the Short Name of an object to be the same as the Title. The difference between the two is that the Short Name is all lower case and hyphenated between all words. For example the Short Name for the page titled About Us would be *about-us*. The Short Name is what Plone also uses in the URL for the page (www.myplonesite.org/about-us). You can specify a different Short Name for an object by using the Rename button on the Contents tab.

Creator

Sorting by the Creator will group all results in alphabetical order by their author. For example, let's say we had several documents published by Bob Baker and several of other documents published by Jane Smith. Sorting by the creator would result in all the documents created by Bob Baker listed first followed by those of Jane Smith.

Title

Sorting by Title will display the results in alphabetical order, by the object titles.

Next we will cover the Dates that we skipped over in this section as well as the Criteria Field section.

Using and Understanding Dates

Explanation of the Dates associated with Collections and their uses

There are several different types of dates we can choose from, many of them sounding similar. Because of this it is very easy to get confused about which date to use. Below, each date option is defined.

Dates Defined

Creation Date The Creation Date is the date the document was made. You can think of this as its birthday, the day it was born. You cannot change the Creation Date of an object.

Effective Date The Effective Date is the date when an object becomes published. This date is customizable through the **Edit tab** on objects under the **Date tab**. However, there it is referred to as the Publishing Date (a minor discrepancy in Plone's nomenclature).

Creation Date and **Effective Date** are very similar. They both are representative of the beginning point of an object. A very important point to keep in mind when choosing which one you want to use, is that an object can be created long before it ever becomes public. You could have a page that is worked on for several weeks before it is actually published. Thus you would get different results in a Collection depending on which date you used. We recommend

using the **Effective Date**, instead of Creation Date for date-oriented Collections. This way your Collection shows results based on when they became viewable to the public, which is more relevant to the audience of your Collection. Also, you can go in and manually adjust the Effective Date to control the sort order which is not something you can do with the Creation Date.

Expiration Date The Expiration Date refers to the day that the item will no longer become publicly available. This date is also customizable through the Edit tab (shown above) like the Effective Date. By default, objects have no Expiration Date.

Modification Date The Modification Date is the date the object was last edited. Note that this date is first set the day the object is created and will automatically change every time the object is edited. There is no way to customize this date. You could use this as a Sort Order along with an Item Type criterion set to Page, to display all recently modified pages within the last week, for example. The What's New listing on the homepage of LearnPlone.Org uses Modification Date as its date criterion. That way both newly created documents *and* ones that have been updated appear in the listing.

Event Specific Dates The two following dates **only** apply to objects that are **Events**. These two dates are very effective for creating Recent Events and Upcoming Events Collections that will let your audience know what your organization is doing and will be doing in the future.

Start Date The Start Date is simply the date that an Event starts.

End Date The End Date is simply the date that the Event ends.

Publication Date

The Publication Date is the date when object was last published. It can either be set manually by means of Effective Date field or, if the latter hasn't been set, calculated based on date when object was last published.

To display Publication Date on your pages you need to switch it on with “*Display publication date in ‘about’ information*” option in **Site Settings Control Panel**. Publication Date will be visible right before object Modification Date inside ‘about’ information area. Make sure “*Allow anyone to view ‘about’ information*” option is also enabled inside **Security Settings Control Panel** to make it all work.

Setting Dates

A confusing thing about dates can be how its Criteria are set up. They have a setup that is not like any of the other Criteria. First off, you have to choose whether you want a Relative Date or a Date Range.

The Relative Date allows you to construct a **conditional statement**. Such as: Items modified less than 5 days in the past. A Date Range will allow you to **specify an exact range of dates**, such as 01/02/08 to 02/02/08. The Date Range is useful when you want to create a Collection with a static date that won't change. The Relative Date can be very useful as it will allow you to create Collections that are automatically updating themselves, such as a Recent News Collections or an Upcoming Event Section.

Relative Date

Looking first at the Relative Date option, you can see we have three options to fill out.

The first option is **Which Day**. This allows us to select the number of days our criterion will include. One of the options is called *Now*. Using this will set the date range to the current day. The other two options do not matter and can be ignored when using *Now*.

The second option is **In the Past or Future**. This enables us to choose whether we are looking forward or backward into time.

The last option is **More or Less**. Here we can choose from three options. *Less than* allows us to include everything from now to a period of time equal to or less than the **Which Day** setting, either in the past or future. *More than* will include everything from beyond our specified number of days equal to or more than **Which Day**. Finally *On the Day*

will only include things that are on the day we specified in the **Which Day**. Using the example in the image above if we had selected *On the Day* instead of *Less than* our Collection would display only objects that were modified (we are using the Modification Date criterion) 5 days ago.

If this is confusing to you, try reading it as a statement substituting in the field options you chose. “I want the results to include objects **More or Less than Which Day, In the Past or Future**”. Our example in the image above would become “I want the results to include objects **Less than 5 days in the past**”.

Date Range

The **Date Range** is much easier to understand. Both a Start Date and End Date are required (do not confuse these terms with the Event Specific dates!). The Date Range allows us to enter a beginning and an end date and the display everything within that time frame. Notice also that it allows us to specify a specific time of day as well.

Portlet Management

An introduction to the use and management of portlets.

Managing Portlets

To assign the different *types of portlet* into your site, use the “Manage portlets” item on the Toolbar.

There are three locations where portlets can be put: to the left, to the right, and in the footer.

Note: The “left”, “right” and “footer” locations come from the classical website design. It is entirely possible, using Diazo theming, to move these portlet locations from one place to another. In fact, many so-called *responsive* designs, that automatically scale for mobile devices, will not display these as ‘left’ and ‘right’.

There is still a relevance to these different locations, even on mobile devices: as a rule of thumb, the ‘left’ portlets will be displayed **before** the main content, and the ‘right’ portlets afterwards, with the ‘footer’ portlets last.

```
*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
```

```
Run keyword if not sys.argv[0].startswith('bin/robot')
...           Setup Plone site  ${FIXTURE}
Run keyword if sys.argv[0].startswith('bin/robot')
...           Open test browser
Run keyword and ignore error Set window size  @${DIMENSIONS}

Test Setup
  Import library Remote  ${PLONE_URL}/RobotRemote

  Run keyword if sys.argv[0].startswith('bin/robot')
  ...           Remote ZODB SetUp  ${FIXTURE}

  ${language} = Get environment variable LANGUAGE 'en'
  Set default language  ${language}

  Enable autologin as Manager
  ${user_id} = Translate user_id
  ... default=jane-doe
  ${user_fullname} = Translate user_fullname
  ... default=Jane Doe
  Create user  ${user_id} Member fullname=${user_fullname}
  Set autologin username  ${user_id}

Test Teardown
  Run keyword if sys.argv[0].startswith('bin/robot')
  ...           Remote ZODB TearDown  ${FIXTURE}

Suite Teardown
  Run keyword if not sys.argv[0].startswith('bin/robot')
  ...           Teardown Plone Site
  Run keyword if sys.argv[0].startswith('bin/robot')
  ...           Close all browsers

*** Test Cases ***

Show portlet management
  Go to  ${PLONE_URL}
  Click link  css=#plone-contentmenu-portletmanager a

  Wait until element is visible
  ...  css=#plone-contentmenu-portletmanager li.plone-toolbar-submenu-header

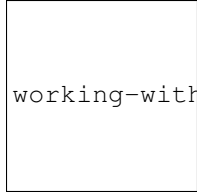
  Update element style  portal-footer  display  none

  Capture and crop page screenshot
  ...  ${CURDIR}/../../_robot/portlet-menu.png
  ...  css=div.plone-toolbar-container
  ...  css=#content-header

Show right portlets
  Go to  ${PLONE_URL}/@@topbar-manage-portlets/plone.footerportlets

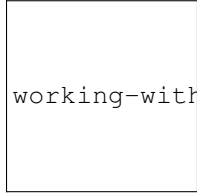
  Capture and crop page screenshot
  ...  ${CURDIR}/../../_robot/portlet-footer.png
  ...  css=#content
```

From here, you choose which region you want to work on. In the example below, we are working on the “footer portlets”



working-with-content/portlet-management/../../../../_robot/port

Note: The footer portlet region is new for Plone 5. If you have worked on previous versions of Plone: this is where you now can find (and edit) the colophon and other items.



working-with-content/portlet-management/../../../../_robot/port

The various options for “blocking” are explained in the *Portlet hierarchy* section.

Adding a Portlet

Adding a Portlet is as simple as selecting the **Add Portlet** drop down box and clicking on the type of Portlet you would like to add. We will cover the different options available in the *next section*.

Editing an Existing Portlet

To edit the properties of an existing Portlet, simply click on the name of the Portlet. If we wanted to edit the properties of the Navigation Portlet, we would Click on *Navigation*. Each type of Portlet will have different configuration options available to it.

Rearranging Portlets

To rearrange your Portlets, simply click the **up or down arrow**. This will affect the order your Portlets are displayed on the page.

Removing Portlets

To remove a Portlet, click the **“X”** associated with its name.

Hiding Portlets

You can show/hide portlets using the associated show/hide link.

Adding multiple portlets

With Portlets, you can add more than one of the same type on a page. There is no limit (except common sense) to how many times you can use an individual Portlet or a limit to how many total Portlets can be on a Page.

Portlet Hierarchy

Portlets use a basic hierarchy approach which determines how and why they appear on each section of your site.

By default, the portlets that you assign at the root (homepage) of the site will propagate down to all the subsections of the site.

If you want a different set of portlets or order of portlets for a particular sub-section, you can use the **Block/unblock portlets** controls, to “block” the parent portlets. When you block Portlets, you must explicitly add all the Portlets that you wish to see on the child page.

There are some specific sets of portlets:

Group Portlets Created by your Site Manager for Groups of Users.

Content Type Portlets Assigned in the Plone Control Panel to specific Content Types by your Site Manager .

They can be blocked or unblocked separately from the (much more used) “Parent portlets”

Note: Beware of the *default view for a Folder*.

When you set about creating Portlets that you want to propagate, be careful to that you are creating the Portlet at the Site Root or Folder and not at the Default View for that folder.

In this diagram, our Portlets are designated in blue underneath the Page title:

As you can see we have two Portlets designated on our Home page (navigation and recent items). These same Portlets appear on our About Page because of portlet hierarchy.

However, on the Documentation page we added a third portlet - the Collection Portlet. Here we are still allowing Parental Portlets, but in addition we specifically added the Collection Portlet.

On **both** the Tutorials and Videos Pages we have to block Parental Portlets because we do not want the Collection Portlet that is on the Documentation Page to show. When we block Parental Portlets we must re-add the Portlets to **each** Child page. In this case we re-added the Navigation Portlet to both and then added the Search Portlet to both.

Remember that the child pages only inherit from the parent page directly above them. In our example, if we added a page called *Staff* under About and allowed the parent portlets without adding any additional portlets, it would show the same Portlets as the Home Page as well as the About Page. However do not think that it is inheriting from the Home page. If we were to change the About Page and added a Search Portlet, our Staff Page would reflect the Portlets on the About Page not the Home Page.

Portlet Types

Descriptions of each Portlet Type

There are several different types of Portlets to choose from.

Note:

These are the default portlets for Plone 5. Your site may have more available, as many add-ons come with their own portlets.

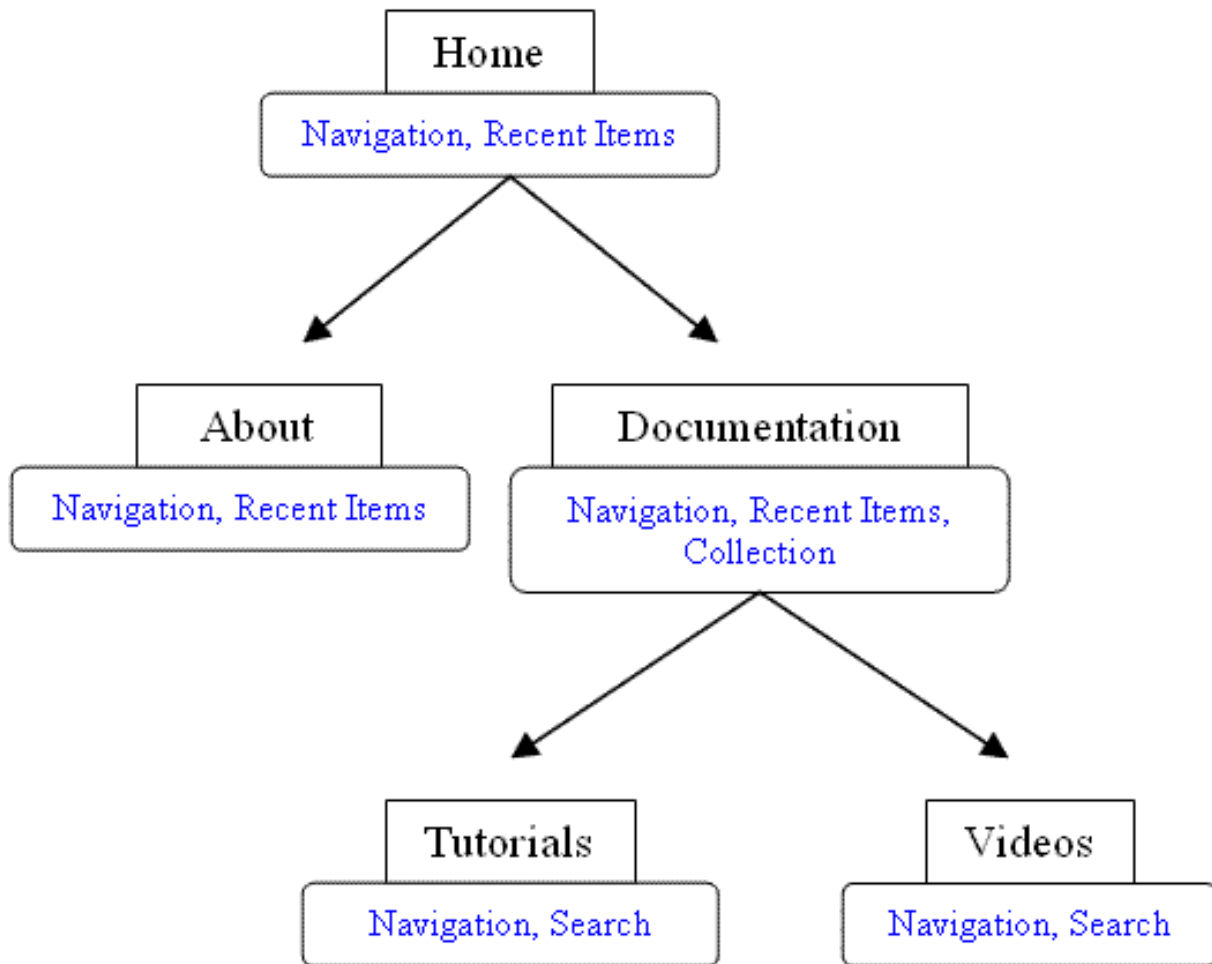


Fig. 3.6: Portlet Hierarchy

Navigation

The Navigation Portlet **allows users to navigate your site** with ease by providing a structured “site map”, or navigation tree.

You have the option to display the navigation for the overall site or choose to display the current folder contents.

As you dig deeper into the site, the tree will continue to expand.

There are several configuration options available that effect how the Navigation Portlet will behave.

You can set the *Title*, the *Root node* (which is from where the tree will start), and various other options to limit how deep you want to expand the tree.

Calendar

The Calendar Portlet is a simple Portlet that will display a Calendar on your site.

The main use is to highlight “Events”.

You can choose what the workflow state of the Events (or other items like it) should be, and can choose whether you want all events on your site or only the ones within a given subfolder.

If you have published Event content objects on your site, the days upon which they occur will be highlighted in the calendar and will link to the corresponding events on your site.

Collection

The Collection Portlet will allow you to **display the results of a Collection**.

You must have a Collection previously created when you add this Portlet, then you can specifying the Collection to be used.

This is a great way to present targeted queries like for example “newest pages with tag “Latin America”.

Events

The Events Portlet will **display Upcoming Events**, provided that you have Events on your site. You can determine how many events you want to be displayed and also which events you want to display based on publishing state.

Log in

The Log in Portlet is another non configurable Portlet that will simply **display a Log in Form** that will allow users with Log in information to log in to the site.

Once a user is logged into the site, this Portlet will not appear.

News

The News Portlet works exactly like the Events Portlet.

Instead of displaying Events, it **displays recent News items**.

Once again you can choose how many News items are displayed and filter them based on their state.

Note: Both the “News” and “Events” portlets could be created by using a “Collection” portlet which finds the right content items.

They are provided for convenience.

RSS Feeds

The RSS Feed Portlet allows you to link to an RSS Feed, choose how many items to display, and specify the refresh rate.

Recent Items

The Recent Items Portlet displays a customizable **number of Recent Items**, listed by Title. A Recent Item is determined by its Last Modified Date.

Review List

The Review List Portlet will display a **list of objects that have been submitted for review**.

If you are using a submit and review cycle (and have properly set global roles for your users) this is a great way for reviewers to see what content has been submitted for review.

This Portlet only appears to those logged in as this state is not viewable to the public.

Search

The Search Portlet will place a search box in your Portlet Column. This search box will search the Titles, Descriptions, and Body text of objects on your site for the text specified.

You have the option of enabling Live Search.

Live Search is a feature which shows live results if the browser supports JavaScript.

Static Text

The Static Text Portlet allows you to enter content just as you would on a normal Page object. This is useful for adding for example contact information, your company motto, or any information that is relatively static.

Classic

A Classic Portlet is refers to the way portlets were used in older version of Plone, prior to Plone 3.

You must create a Page Template in the Management Interface and properly set the path and macro to enable the portlet.

This requires technical knowledge of both TALES and the Management Interface.

Warning: The ‘classic’ portlet is provided strictly for sites that have very old legacy code.

You should refrain from using it in any modern Plone site.

Create and maintain good quality content

Description

Plone comes with several tools to maintain the quality of the content you create. This ensures both good results in search engines, as better usability for the visitors.

What is “content quality”?

Under this heading we list several tools that help you to ensure content can be properly indexed, is enriched with meta-data such as tags, dates and even geo-location, and to make sure links are not broken.

This will also help you with accessibility certification for your content, and Search Engine Optimization (SEO)

Batteries included

Plone comes standard with a whole host of features that help you:

- The URL’s for content are derived from the Title, making sure you have human-readable URL’s
- [Dublin Core](#) metadata is used throughout
- The navigation is automatic, and within folders you can also enable “previous/next” style links
- Automatic filters in the editor ensure that the page will be saved as valid HTML
- Behind the scenes, Plone registers internal links with so-called “UUIDs”. In short, a unique key is generated for every content item, making sure all internal links will work if you move pieces of content or even whole folders around.
- When you delete a piece of content, you will receive a warning if there are other places in your site that still link to this content. You’ll get the option to correct those other pages.
- If you move content around and people come to your site using the old URL, they will be automatically redirected to the new location. (*Tip: you can even use this to create short ‘alias’ URL’s...*)
- Images are automatically scaled. Even if your editors upload high-resolution images, you will get smaller sizes that ensure quick loading of your page. Of course the original image size is also available, if you want.

And there are some hidden gems as well: you can enable (in the Control Panel, as site administrator) the [After The Deadline](#) intelligent spelling and grammar checker. Now while this is only available at the moment for English, French, Spanish and Portuguese, if your site uses one of those languages this is a very valuable add-on. For testing and light use, you can use the connection to the online service; if you have many users or create much content, you are advised to set up your own instance of the After The Deadline server. It is free and open source software.

In the next section, we will point to several add-ons that can help even more to create and maintain high-quality content.

Content Quality helper tools

Description

A selection of add-ons that can help create and maintain appealing, searchable, and high-quality content.

Apart from the *inbuilt tools*, there are several add-ons available.

Note that these are all separate add-ons you will have to *install*, and we strongly suggest testing them out first on a separate test-instance of your site, to see if they fit your purpose and do not interfere with other parts of your site. Also, some of these tools rely on web-services, which may or may not be allowed or advisable in high-security scenarios.

Avoiding content errors

- [collective.jekyll](#) is a package that will help you identify common pitfalls, like too long or short titles or descriptions, or a URL starting with “copy_of”. You can even set it up so it alerts editors when they don’t stick to the preferred image format, or if a page has not enough links to other pages.
- [eea.progressbar](#) can provide a visual clue as to where a document is in the workflow progress, making it easier for editors and reviewers to track what to do next to publish a document.

Check your links

- [collective.linkcheck](#) provides link validity checking and reporting.
- although you may also want to keep this out of Plone itself, and run an external linkchecker regularly. [This Linkchecker](#) is open source, available for multiple platforms and can be scripted.

Better images

- [Products.ImageEditor](#) allows you to rotate, flip, blur, compress, change contrast & brightness, sharpen, add drop shadows, crop, resize an image, and apply sepia.
- [collective.aviary](#) integrates the external “Aviary” image editor into Plone.
- [plone.app.imagecropping](#) surprisingly enough, crops images.

Tags, relations and more

- [eea.tags](#) provides a Facebook-like autocomplete widget for tagging content.
- [eea.alchemy](#) allows you to bulk auto-discover geographical coverage, temporal coverage, keywords and more.
- [collective.taghelper](#) can connect to a range of webservices to assist tagging
- [collective.simserver](#) can help with creating ‘related items’ links
- [collective.taxonomy](#) can set up hierarchical taxonomies in multiple languages
- [collective.classifiers](#) provides a ‘middle ground’ between a complex taxonomy and simple tagging, allowing for two new fields to classify content
- [collective.facets](#) is an alternative approach allowing editors to add ‘facets’ to content.

Analytics and SEO

- [collective.googleanalytics](#) enables easy tracking of the standard Google statistics as well as external links, e-mail address clicks and file downloads. It also defines Analytics reports that are used to query Google and display the results using Google Visualizations.
- [quintagroup.seoptimizer](#) allows setting various meta tags and other information search engines like and need.

- if you have migrated from another system, and need to set up aliases to content that still lives in search engines, [Products.RedirectionTool](#) gives you an interface to Plone's built-in redirection and aliasing.

And after all that work, you can use [quintagroup.analytics](#) to see your webmaster stats increase. Now lean back with your favorite hot beverage, you've earned it!

Adapting & Extending Plone

Basic Changes (Look and Feel)

Change the Logo

```
*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Open test browser
    Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
    Import library  Remote  ${PLONE_URL}/RobotRemote
```

```
Run keyword if sys.argv[0].startswith('bin/robot')
...           Remote ZODB SetUp  ${FIXTURE}

${language} = Get environment variable  LANGUAGE  'en'
Set default language  ${language}

Enable autologin as  Manager
${user_id} = Translate  user_id
... default=jane-doe
${user_fullname} = Translate  user_fullname
... default=Jane Doe
Create user  ${user_id}  Member  fullname=${user_fullname}
Set autologin username  ${user_id}

Test Teardown
Run keyword if sys.argv[0].startswith('bin/robot')
...           Remote ZODB TearDown  ${FIXTURE}

Suite Teardown
Run keyword if not sys.argv[0].startswith('bin/robot')
...           Teardown Plone Site
Run keyword if sys.argv[0].startswith('bin/robot')
...           Close all browsers
```

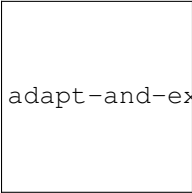
How to substitute the standard Plone logo with your own logo - a through-the-web approach.

The Basics

In Plone 5, the logo can be changed TTW in the @@site-controlpanel.

1. Changing the Image in the Site control panel

Since Plone 5 you can directly change the logo in the Site control panel. Just upload your custom logo image with the Site logo field.



adapt-and-extend/../../_robot/change-logo-in-site-control-panel

```
*** Keywords ***

Highlight field
[Arguments]  ${locator}
Update element style  ${locator}  padding  0.5em
Highlight  ${locator}

*** Test Cases ***

Take annotated screenshot
Go to  ${PLONE_URL}/@@site-controlpanel
```

```
Highlight field  css=#formfield-form-widgets-site_logo
Capture and crop page screenshot
...    ${CURDIR}/../_robot/change-logo-in-site-control-panel.png
...    css=#content
...    css=#formfield-form-widgets-site_logo
```

3. Changing the HTML

To change the HTML of the logo part you can use Diazo to just copy the src and href of the logo elements and put them in your custom HTML in your static HTML Theme. For further information's about Diazo please have a look at the Diazo documentation in [Theming Plone](#).

Further Information

- There are further How Tos in the Logo section of the Plone documentation dealing with more advanced customization methods.
- More guidance on TAL and ZPT can be found in the ZPT tutorial.
- If you want to transfer your changes to the file system in your own theme product, then proceed to the [viewlets overview section](#).

Change the Font Colors - deprecated

Note: This tutorial is deprecated for Plone 5. New documenation is almost ready.

How to change the font colors - a through-the-web approach.

You'll be introduced to some very simple techniques here for through-the-web customizations of Plone's CSS.

- How to locate the styles you want to change
- How to override these styles using the ploneCustom.css style sheet

In this case we'll change the color of page titles from black to turquoise.

Before you start

For convenience, Plone themes often comprise a number of separate style sheets, but for speed and efficiency, in production mode, Plone has a mechanism (portal_css) for packaging these up into just one or two files.

You'll need to disable this when making changes to your site or customizing CSS. So make sure you've followed the instructions on how to put your site into [debug mode](#).

Locating the styles you want to change

- If you don't already have a page in your Plone site, add one, save it and inspect it in view mode.
- Use [Firebug](#) , or a similar tool, to locate the class name of the page title - in this case its h1.documentFirstHeading.

Locating the ploneCustom.css style sheet

As a matter of course, the last style sheet to load on every Plone page is `ploneCustom.css`. You'll see this if you inspect the HTML head tag of your page using Firebug. If you dig further, you'll probably find that this style sheet is completely empty. By the rules of precedence in the CSS Cascade, any styles in this sheet will override styles specified in the preceding sheets. So you have a "blank sheet" here for your own customizations.

The trick now is to locate this file, so that you can make it available for editing.

To make life easier for yourself, you might like to open a second tab or browser window at this point - you can then quickly switch back to the first tab to see your changes.

Go to Site Setup > Management Interface and click `portal_skins`

Use the Find option in the tabs across the top to locate `ploneCustom.css`:

- Type `ploneCustom.css` in the "with ids:" box and click Find
- You may get more than one result, it doesn't matter which you choose to click on, however best practice is to choose the one flagged with the red asterisk.

Customizing and Editing ploneCustom.css

When you click on `ploneCustom.css` you'll find that you can't edit it. The next stage is to put the `ploneCustom.css` in a place where it can be edited. You'll see a Customize option just above the grey text area, click the Customize button and you'll find that the style sheet has been automatically copied to `portal_skins/custom`.

You're now free to edit the file as you like. To change the color of our page titles, add:

```
h1.documentFirstHeading {  
    color: #0AAE95;  
}
```

and save.

If you've installed Plone 4 with the Sunburst theme, the `ploneCustom.css` comes with a number of commented out pre-packaged styles that you might like to experiment with. You can override the layout styles to a fixed width and alter the colors of the links.

Rolling back your changes

You've got a couple of options for reverting back to the original CSS:

comment out your styles in the `ploneCustom.css` - the usual CSS commenting syntax applies

delete (or, if you want to keep a note of what you did, rename) your version of `ploneCustom.css`, you'll find it here:

- Site Setup > Management Interface > `portal_skins` > `custom`
- you can choose the delete or rename options - try renaming to `ploneCustom.css.old`
- you can then go back to the beginning of the process of locating and customizing `ploneCustom.css`

Further Information

You've actually encountered two types of customization here.

1. The first is a standard method of using order of precedence - the Cascade - to overwrite CSS styles as they reach the browser.

2. The second is a Plone/Zope specific method of overriding the style sheets themselves by dropping them into the custom folder of `portal_skins`. This method can also be used for templates and other resources and is explained in more *depth in the section on Skin Layers* in this manual.

More advanced techniques, including incorporating your own style sheets into a theme product, are covered later in this manual.

You can find out more about how the CSS Registry (`portal_css`) packages up the style sheets to deliver them to the page in the *Templates and Components to Page* section of this manual.

Remove “Welcome!” section from splash page

Two methods to remove the auto-generated Welcome! section from root splash page.

The Basics

In Plone 5, an auto-generated Welcome! section is created on a root splash page if its short name is “front-page”. This happens because of a [Diazo theme rule](#) tied with a condition to Plone’s default home page, something like this:

```
<!-- include view @@hero on homepage only -->
<after css:theme="#mainnavigation-wrapper" css:content=".principal" href="/@@hero"
      css:if-content="body.template-document_view.section-front-page" />
```

Two ways to remove this section include:

1. Creating a new splash page with different name

1. Simply add a new page in the root folder and give it a short name other than “front-page”. Easier yet: you can just rename default homepage’s shortname with `Edit > Settings`.
2. Upon saving this new page (or renaming old one), you can then press “Display” from the toolbar and select “Change content item as default view...”
3. Finally select the radio button of your new page and press Save.

2. Duplicating theme and removing line of code

1. Go to site setup
2. Press “Theming”
3. Press “Copy” on the Barceloneta theme, give it any short title and description and you can activate it now or later. Press Create.
4. Click on “rules.xml” to open this file in the browser.
5. Find and delete the rule mentioned above: `<after css:theme="#mainnavigation-wrapper" css:content=".principal" href="/@@hero" css:if-content="body.template-document_view.section-front-page" />`

Then to finish up...

1. Press the blue “Save” button, then press “Back to control panel”.
2. If you’ve not already activated it, press “Activate”, then “Clear Cache” on your new theme.

Theming Plone

Intro

The current best-practice way to theme a Plone site is by using an engine called “**Diazo**”. This allows designers to design a theme in just plain, flat HTML, CSS (and JavaScript, if wanted) and then to hook that into the Plone backend to fill it with sophisticated content.

The easiest way to do this is to use “`plone.app.theming`”. But if you need to integrate Plone with other back-end servers, legacy systems, or any webservice, you can use Diazo to all combine it in a unified look & feel.

Create a Plone 5 theme product (addon)

Introduction

Creating a theme product with the Diazo inline editor is an easy way to start and to test, but it is not a solid long term solution.

Even if `plone.app.theming` allows to import and export a Diazo theme as a ZIP archive, it might be preferable to manage your theme in an actual Plone product.

One of the most obvious reason is it will allow you to override Plone elements that are not accessible from the pure Diazo features (like overloading content views templates, viewlets, configuration settings, etc.).

Create a product to handle your Diazo theme

To create a Plone 5 theme skeleton, you will use mrbob’s templates for Plone.

Install mr.bob and bobtemplates.plone

To install mr.bob you can do:

```
$ pip install mr.bob
```

and to install the needed bobtemplates for Plone, do:

```
$ pip install bobtemplates.plone
```

Create a Plone 5 theme product skeleton with mrbob:

```
$ mrbob -O plonetheme.tango bobtemplates:plone_addon
```

It will ask you some questions:

```
--> What kind of package would you like to create? Choose between 'Basic', 'Dexterity
↪', and 'Theme'. [Basic]: Theme
```

here choose Theme and fill out the rest of the questions as you like:


```
--> Author's name [MrTango]:
--> Author's email [md@derico.de]:
--> Author's github username: MrTango
--> Package description [An add-on for Plone]: Plone theme tango
--> Plone version [4.3.6]: 5.0b3

Generated file structure at /home/maik/develop/plone/plonetheme.tango
```

Now you have a new python package in your current folder:

```
(mrbob)maik@planetmobile:~/develop/plone/plonetheme.tango
$ ls
bootstrap-buildout.py  buildout.cfg  CONTRIBUTORS.rst  MANIFEST.in  setup.py  travis.
↳cfg
bootstrap-buildout.pyc  CHANGES.rst  docs  README.rst  src
```

You can run:

```
$ python bootstrap-buildout.py
Creating directory '/home/maik/develop/plone/plonetheme.tango/bin'.
Creating directory '/home/maik/develop/plone/plonetheme.tango/parts'.
Creating directory '/home/maik/develop/plone/plonetheme.tango/develop-eggs'.
Generated script '/home/maik/develop/plone/plonetheme.tango/bin/buildout'.
```

Then you can run:

```
$ ./bin/buildout
```

This will create the whole development environment for your package:

```
$ ls bin/
buildout                                code-analysis-hasattr          develop  ↳
↳pildriver.py                          code-analysis-imports          flake8   ↳
↳pilfile.py                            code-analysis-jscs             fullrelease ↳
↳pilfont.py                            code-analysis-jshint            instance  ↳
↳pilprint.py                           code-analysis-pep3101          lasttagdiff ↳
↳postrelease                           code-analysis-prefer-single-quotes lasttaglog ↳
↳prerelease                            code-analysis-utf8-header       longestest ↳
↳release                               code-analysis-zptlint          pilconvert.py ↳
↳test
```

You can run:

```
$ ./bin/instance fg
```

to start a Plone instance and play with your packaged.

Your package source code is in the src folder:

```
$ tree src/plonetheme/tango/
src/plonetheme/tango/
- browser
|   - configure.zcml
|   - __init__.py
|   - __init__.pyc
|   - overrides
|   - static
- configure.zcml
- __init__.py
- interfaces.py
- locales
|   - plonetheme.tango.pot
|   - update.sh
- profiles
|   - default
|       |   - browserlayer.xml
|       |   - metadata.xml
|       |   - plonethemetango_default.txt
|       |   - theme.xml
|   - uninstall
|       |   - browserlayer.xml
|       |   - plonethemetango_uninstall.txt
|       |   - theme.xml
- setuphandlers.py
- testing.py
- tests
|   - __init__.py
|   - __init__.pyc
|   - robot
|       |   - test_example.robot
|   - test_robot.py
|   - test_setup.py
- theme
    - index.html
    - manifest.cfg
    - rules.xml
    - template-overrides

11 directories, 25 files
```

As you see, the packages contains already a Diazo theme:

```
$ tree src/plonetheme/tango/theme/
src/plonetheme/tango/theme/
- index.html
- manifest.cfg
- rules.xml
- template-overrides
```

Here you can build your Diazo theme. For details how to do that, look at [plone.app.theming](#) and [Diazo](#).

Override Plone BrowserViews with jbot

A large part of the Plone UI is provided by [BrowserView](#) or [Viewlet](#) templates.

That is the case for viewlets (all the blocks you can see when you call the url `./@@manage-viewlets`).

Note: To override them from the Management Interface, you can go to `./portal_view_customizations`.

To override them from your theme product, the easiest way is to use `z3c.jbot` (Just a Bunch of Templates).

Since `jbot` is already included in the skeleton, you can just start using it, by putting in `src/plonetheme/tango/browser/overrides/` all the templates you want to override. But you will need to name them by prefixing the template name by its complete path to its original version.

For instance, to override `colophon.pt` from `plone.app.layout`, knowing this template in a subfolder named `viewlets`, you need to name it `plone.app.layout.viewlets.colophon.pt`.

Note: Management Interface > `portal_view_customizations` is a handy way to find the template path.

You can now restart Zope and re-install your product from the Plone control panel (Site Setup > Add-ons).

Plone 5 Resource Registry

Description

Plone 5 has modernized the JavaScript and CSS development experience by incorporating tools like Bower, Grunt, RequireJS and Less. JavaScript and CSS resources for core Plone and add-on packages are managed in the new Plone 5 Resource Registry. The Resource Registry was completely rewritten in Plone 5 to support a new dependency-based approach built on RequireJS. It also allows developers to create JavaScript and CSS bundles Through-The-Web for a low barrier to entry. This chapter will help you to gain a basic understanding of the new Resource Registry.

Introduction to Plone 5 resources

Plone 5 introduces new concepts for working with JavaScript and CSS in Plone.

JavaScript and AMD

Prior to Plone 5, JavaScript resources in Plone have been registered with the `portal_javascripts` tool using the Generic Setup import step `jsregistry.xml`. This approach allowed a developer to specify the order in which individual JavaScript resources were loaded into an HTML page, and controlled compiling and minifying these resources for production into a minimal set of files. As the use of JavaScript in front-end development expanded, this model proved insufficient to solve complex dependency management issues. At the same time, developments in the JavaScript community led to a new approach to handling complex dependencies.

Starting in Plone 5, we use a new [module](#)-based solution on the [Asynchronous Module Definition](#) (AMD) approach. The new Plone 5 Resource Registry uses [RequireJS](#) to allow developers to define individual JavaScript modules and the dependencies they will require. Because RequireJS uses the AMD approach, these modules and their dependencies can be served in an uncompiled form during development and then compiled and minified Through-The-Web for use in production. This allows a “development mode” where changes in JavaScript source files are reflected in the browser in near real-time.

It is also possible in the new Resource Registry to provide bundles with simple JavaScript that does not make use of AMD. See [Non-AMD Bundles](#) below for an example of such a bundle.

CSS and Less

Modern web development relies on CSS (Cascading Style Sheets). In order to support complex CSS, “preprocessors” have developed that allow a more programmatic way of defining styles and sharing common elements. Plone 5 has chosen to use the [Less CSS preprocessor](#) because it is compiled by JavaScript tools, which fit with the new Plone Resource Registry. Less provides nice features like inheritance, scoping, functions, mixins and variables, which are not available in pure CSS.

Resources and Bundles

In the Plone 5 Resource Registry, JavaScript and CSS are organized into *resources* and *bundles*. A resource consists of one JavaScript and either none, one or multiple Less or CSS files. A bundle combines several resources into a single unit that is used in a webpage. In “development mode” the resources for a bundle may be served individually and unminified to facilitate development. In “production mode” a bundle’s resources are compiled and minified to minimize the number of requests needed to deliver the bundle to a client.

In the sections below we will discuss *resources* and *bundles* in depth.

Resources

Resources are the main unit of the resource registry. A resource may contain at most one JavaScript file and zero or more CSS/Less files. RequireJS identifies resources by name.

Resources - as well as *bundles* - are registered with a `records` element in the `registry.xml` Generic Setup import step. The Plone 5 Resource Registry reads these `records` and builds RequireJS configuration for compiling resources into *bundles*.

A resource record element must have two attributes: “prefix” and “interface”. The value of “interface” must be `Products.CMFPlone.interfaces.IResourceRegistry`. The value of “prefix” must begin with `plone.resources/` followed by a unique value that will be used by RequireJS as the **name** of the resource. To ensure that your bundle has a unique name, we suggest that you use the name of your package. You should convert dots to dashes to conform to RequireJS naming standards. See below for *some examples* of different types of resource records.

JavaScript resources registered should conform to the RequireJS module pattern. However, we can also include non-module, legacy resources which do not make use of the RequireJS `define` and `require` methods. If you must register such resources, use the *shim options* defined below.

Resource Options

Options are defined on a resource record using `value` elements in the form `<value name="option_name">option_value</value>`. The options that may be used on any resource record are:

js URL of the JavaScript file.

css URLs of CSS/Less elements.

url Base URL for loading additional resources like text files. See below for *an example*.

For these options, the URL you provide as a value must point to a file in a *resource folder*. Optionally, you may choose to register a directory in your package using the `++plone++static` traversal namespace.

Shim Resources

If the JavaScript you wish to register does not follow the RequireJS module pattern (using `define` and `require`), you may still register it in a resource. You will need to use the `shim` options for your resource record. We refer to this kind of JavaScript as “legacy”, as it doesn’t follow our proposed best practices. For more information on configuring shims in RequireJS, see: <http://requirejs.org/docs/api.html#config-shim>

export Shim export option to define a global variable where the JavaScript module should be made available.

deps Shim depends option to define which other RequireJS resources should be loaded before this shim module.

init Shim init option to define some JavaScript code to be run on initialization.

Example Resource Records

Here are some examples of different types of resource records (all examples below are from `Products.CMFPlone`).

An example of a resource record for a single javascript module:

```
<records prefix="plone.resources/mockup-router"
  interface='Products.CMFPlone.interfaces.IResourceRegistry'>
  <value key="js">++resource++mockupjs/router.js</value>
</records>
```

An example of a resource record for a single Less file:

```
<records prefix="plone.resources/bootstrap-variables"
  interface='Products.CMFPlone.interfaces.IResourceRegistry'>
  <value key="css">
    <element>++plone++static/components/bootstrap/less/variables.less</element>
  </value>
</records>
```

An example of a resource for multiple Less files:

```
<records prefix="plone.resources/bootstrap-basic"
  interface='Products.CMFPlone.interfaces.IResourceRegistry'>
  <value key="css">
    <element>++plone++static/components/bootstrap/less/utilities.less</element>
    <element>++plone++static/components/bootstrap/less/forms.less</element>
    <element>++plone++static/components/bootstrap/less/navs.less</element>
    <element>++plone++static/components/bootstrap/less/navbar.less</element>
  </value>
</records>
```

An example of a resource combining JavaScript and Less/CSS:

```
<records prefix="plone.resources/picker.date"
  interface='Products.CMFPlone.interfaces.IResourceRegistry'>
  <value key="js">++plone++static/components/pickadate/lib/picker.date.js</value>
  <value key="css">
    <element>++plone++static/components/pickadate/lib/themes/classic.date.css</
    <element>
  </value>
  <value key="deps">picker</value>
</records>
```

Note: Please note that because a resource may contain at most one JavaScript file, the url for that file is placed directly into the `<value key="js" />` option. However, as a resource may contain any number of CSS/Less files, each url must be added to the `<value key="css" />` in an `<element />` tag.

The URL Resource Option

The URL option allows you to define the base url for loading other resources needed by your JavaScript.

In the following example from the `mockup` package, the `url` option is used to register a URL base from which an XML template may be loaded. The name of the resource is set as `mockup-patterns-structure`.

In the resource is register in `registry.xml` (from `Products.CMFPlone`):

```
<records prefix="plone.resources/mockup-patterns-structure"
  interface='Products.CMFPlone.interfaces.IResourceRegistry'>
  <value key="js">++resource++mockup/structure/pattern.js</value>
  <value key="url">++resource++mockup/structure</value>
  <value key="css">
    <element>++resource++mockup/structure/less/pattern.structure.less</element>
  </value>
</records>
```

Then in `mockup/configure.zcml` we register a resource directory called `mockup`. The resource traversal namespace `++resource++mockup` points to the filesystem directory `mockup/patterns`.

```
<browser:resourceDirectory
  name="mockup"
  directory="./patterns" />
```

Finally, in `mockup/patterns/structure/js/views/actionmenu.js`, we can list a [text dependency](#). The url base for the dependency is listed as `mockup-patterns-structure-url`. The path that follows will be resolved from the registered resource directory set in the URL option for this resource record: `mockup/patterns/structure`.

```
define([
  'jquery',
  'underscore',
  'backbone',
  'mockup-ui-url/views/base',
  'mockup-utils',
  'text!mockup-patterns-structure-url/templates/actionmenu.xml',
  'bootstrap-dropdown'
], function($, _, Backbone, BaseView, utils, ActionMenuTemplate) {
  'use strict';

  var ActionMenu = BaseView.extend({
    className: 'btn-group actionmenu',
    template: _.template(ActionMenuTemplate),

    // ...
  });
  return ActionMenu;
});
```

Shim Resource Examples

Here is an example of a resource record using shim options (from `Products.CMFPlone.profiles.dependencies`). Here, the variable `tinyMCE` is exported as an attribute of `window`, the global JavaScript namespace. The `init` option is used to define a simple function that will be executed when the `tinymce.js` JavaScript file has been loaded.

TODO: Verify that the above description is true.

```
<records prefix="plone.resources/tinymce"
  interface='Products.CMFPlone.interfaces.IResourceRegistry'>
  <value key="js">++plone++static/components/tinymce/tinymce.js</value>
  <value key="export">window.tinyMCE</value>
  <value key="init">function() { this.tinyMCE.DOM.events.domLoaded = true; return_
  ↪this.tinyMCE; }</value>
  <value key="css">
    <element>++plone++static/components/tinymce/skins/lightgray/skin.min.css</element>
    <element>++plone++static/components/tinymce/skins/lightgray/content.inline.min.css
  ↪</element>
  </value>
</records>
```

In this example, we configure the shim for the backbone resource. This resource exports the backbone javascript library as the `Backbone` attribute of `window`, the global JavaScript namespace. The `deps` option is used to list two resources required by backbone: `underscore` and `jquery`. Note that the format for `deps` is a comma-separated list of resource names. All resources named in `deps` must also be registered with the Plone 5 Resource Registry.

```
<records prefix="plone.resources/backbone"
  interface='Products.CMFPlone.interfaces.IResourceRegistry'>
  <value key="js">++plone++static/components/backbone/backbone.js</value>
  <value key="export">window.Backbone</value>
  <value key="deps">underscore, jquery</value>
</records>
```

Default resources in Plone

Plone 5 ships with a list of Mockup and Bower components for Plone 5's new UI. These resources can be found in the static folder (`Products.CMFPlone.static`), where you can also find the `bower.json` file. These resources are preconfigured in the registry (`registry.xml` in `Products.CMFPlone.profiles.dependencies`).

The ++plone++ traversal namespace

We have a new `plone.resource` based traversal namespace called `++plone++`. Plone registers the `Products.CMFPlone.static` folder for this traversal namespace. Resource contained in this namespace can be stored in the ZODB (where they are looked up first, by default) or in the filesystem. This allows us to customize filesystem based resources Through-The-Web. One advantage of this new namespace over the existing `++resource++` and `++theme++` namespaces is that you may override resources in this namespace one file at a time, rather than needing to override the entire directory.

You may configure a folder in your add-on package to be included in this namespace. To configure a directory in your package, add the following ZCML:

```
<plone:static
  directory="static"
```

```
type="plone"
name="myresources"
/>
```

Now we can access the contents of the “static” folder in your package by using a URL that starts with `++plone++myresources/`. Additional path segments in your URL will be resolved within the “static” folder in your package. For example, `++plone++myresources/js/my-package.js` will correspond to `static/js/my-package.js` within your package.

Note: When providing static resources (JavaScript/Less/CSS) for Plone 5’s resource registry, use `plone.resource` based resources instead of Zope’s browser resources. The latter are cached heavily and you won’t get your changes compiled into bundles, even after Zope restarts.

Bundles

A bundle combines multiple *resources* into a single unit, identified by a name. Bundles can be used to group resources for different purposes. For example, the “plone” bundle provides resources that could be of use to any client, but the “plone-logged-in” bundle supplies resources needed only for those who are logged in to the Plone site.

Generally speaking, when a Plone page is delivered to a client, only bundles will be loaded. There are exceptions, you can register individual resources to be loaded for a specific request via an API method. We will discuss this *a bit later*.

Like *resources*, bundles are registered with a `records` element in the `registry.xml` Generic Setup import step. A bundle record element must have two attributes: “prefix” and “interface”. The value of “interface” must be `Products.CMFPlone.interfaces.IBundleRegistry`. The value of “prefix” must begin with `plone.bundles/` followed by a unique value that will be used as the name of the bundle. See below for *some examples* of different types of resource records.

When developing an add-on you will create your own bundle. Your bundle should include all resources required for your JavaScript or CSS/Less to work properly.

If your bundle will be used only on a single page, you can define it to include it only there. You can use the “expression” option to control including an enabled bundle. You can also use API methods from `Products.CMFPlone.resources` to add disabled bundles to a single request. For example, the `resourceregistry` bundle is only used for the `@@resourceregistry-controlpanel` view. (see the section *Controlling Resource and Bundle Rendering* for more information)

Note: A bundle can depend on other bundles. Declaring such a dependency only controls the order in which bundles are loaded and is mostly relevant for legacy bundles. Currently, bundle dependencies don’t make use of RequireJS dependencies or AMD. Each bundle will be compiled with all dependencies, even if a dependency was already used for another bundle. This raises the response payload unnecessarily.

To avoid this, use the `stub_js_modules` option for your bundle record listed in *Bundle Options* below.

Development vs. Production Mode

In development mode, each bundle loads all of its resources individually. This allows modifications to resources to be immediately available. You do not need to compile any bundles beforehand. You should be aware that this feature does lead to a lot of requests and slow response times, even though RequireJS loads dependencies asynchronously.

In production environments you will compile your bundles to combine and minify all the necessary resources into a single JavaScript and CSS file. Since the dependencies of each resource in the bundle are all now well-defined, they

can all be included in these files. Compiling bundles minimizes the number of web requests and the payload of data sent over the network. In Production mode, only one or two files are included in the output for each active bundle: a JavaScript and a CSS file.

Bundle Options

Options are defined on a bundle record using value elements in the form `<value name="option_name">option_value</value>`. The possible options for a bundle are:

enabled Enable or disable the bundle.

depends List other bundles as dependencies of this bundle. Currently used for the order of inclusion in the rendered content. The defined bundle will only be included in a page after any bundles listed.

resources List the resources that are included in this bundle.

compile Set the value to `True` or `False`. Your bundle must be compiled if it has any Less or RequireJS resources. If you wish, you may precompile your bundles using command line tools provided by Plone or your own preferred toolchain. For more information, *see below*.

If this value is `False`, no button will be provided to compile this bundle Through-The-Web (eg. used for the `plone-legacy bundle`).

expression A TALES expression. If the expression evaluates as `True`, the bundle will be included.

merge_with Indicate in which of the bundle aggregations this bundle should be included. The valid values for this option are `default` or `logged-in`. (*see below*).

conditionalcomment Provide a conditional comment for Internet Explorer hacks.

stub_js_modules Provide a list of resources that are required by this bundle, but already provided by another active bundle. This prevents the stub module from being included multiple times and can reduce the download size of bundles.

New in version 5.0.1.

The following options are used when you provide a pre-compiled bundle. The values will be automatically set when the bundle is built Through-The-Web. If you use the `plone-compile-resources` script, or your own custom toolchain to compile your own bundle JS or CSS, you will need to manage these values yourself.

jscompilation URL of the compiled and minified JavaScript file.

csscompilation URL of the compiled and minified CSS file.

last_compilation Date of the last compilation time. The value of this option is automatically used as version parameter for cache-busting in production mode. (eg. `plone-logged-in-compiled.min.js?version=2015-05-07%2000:00:00.000003`)

Compiling Bundles

There are three ways to provide a compiled version of a bundle for production:

Compile the bundle Through-The-Web and store it in the ZODB

When using this option, all an add-on developer or an integrator needs to do is register a bundle with the “compile” option set to `True`. In the Plone 5 Resource Registry control panel, a button will be available to compile the bundle. Pressing this button will compile the bundle and store it for production delivery.

Compile the bundle from the command line:

Plone provides a script which will compile a specific bundle available in the resource registry. To use this option, you must specifically request the script in your buildout. Add a new part called “resources” and list it in your buildout “parts”, then re-run buildout. You will find the `plone-compile-resources` script in your buildout `bin` directory.

```
[resources]
recipe = zc.recipe.egg
eggs = Products.CMFPlone
scripts = plone-compile-resources
```

Once the script has been created you may invoke it. You will need to provide options indicating the ID of the Plone site in which your package is installed, and the name of the bundle you wish to compile:

```
./bin/plone-compile-resources --site-id=myplonesite --bundle=mybundle
```

This script will start up your Plone site, extract the required information and compile the bundle. Because of this, you will need to stop a Plone instance before running this script.

Use your own compilation chain

The Plone 5 Resource Registry can be used with your favorite build system. Use the tool you prefer create a compiled version of your bundle. Your bundle registration must provide a URL for the “jscompilation” and “csscompilation” options. Your compiled files must be in the filesystem locations that are indicated by these values.

Default Plone bundles

There are three main bundles defined by Plone:

plone: This is the main compiled bundle with all the JavaScript and CSS components required for the Plone Toolbar and the main Mockup patterns.

plone-logged-in: This bundle is only included for logged in users. It contains patterns like the “tinymce” pattern, the “querystring” pattern for collection edit forms and others.

plone-legacy: This bundle is not compiled and contains code that doesn’t use RequireJS or Less. Addons which continue to install resources to `portal_javascripts` or `portal_css` are registered as resources in the plone-legacy bundle automatically.

The legacy bundle

The legacy bundle exists to support packages with code that does not work with the new Plone 5 Resource Registry. Code that cannot be migrated to use RequireJS can be included in the legacy bundle. Code that uses RequireJS in a way which is incompatible with Plone’s use of it (e.g. it’s using its own RequireJS setup) can be included in the legacy bundle.

Note: Some JavaScript use its own setup of RequireJS. Others - like Leaflet 0.7 or DataTables 1.10 - try to register themselves for RequireJS. This can lead to the infamous “mismatched anonymous define” errors (*see below*). You can register such scripts in the `plone-legacy` bundle by including them in the `jsregistry.xml` import step. The `define` and `require` methods are unset before these scripts are included in the output and reset again after all scripts have been included. See yourself: <https://github.com/plone/Products.CMFPlone/pull/870/files>

Resources which are registered into `portal_javascripts` or `portal_css` registries via an addon are automatically registered in the legacy bundle and cleared from `portal_javascripts` and `portal_css`.

Note: JavaScript which doesn't use RequireJS can still be managed by it by including it as a resource with configured shim options.

The plone-legacy bundle treats resources differently: they are not compiled, but simply concatenated and minified.

Example Bundle Records

Here are some examples of Bundle records from Plone and popular add-ons

The record for Plone's plone bundle names a single resource, plone. This is a good example of using a single resource with a require call to bundle a number of other resources, many of which use define, in order to avoid *The mismatched anonymous define error*. (see `Products/CMFPlone/profiles/dependencies/registry.xml` and `Products/CMFPlone/static/plone.js`, and for an example of the bundled resources `mockup/patterns/autotoc/pattern.js`)

```
<records prefix="plone.bundles/plone"
        interface='Products.CMFPlone.interfaces.IBundleRegistry'>
  <value key="resources">
    <element>plone</element>
  </value>
  <value key="enabled">True</value>
  <value key="jscompilation">++plone++static/plone-compiled.js</value>
  <value key="csscompilation">++plone++static/plone-compiled.css</value>
  <value key="last_compilation">2014-08-14 00:00:00</value>
</records>
```

The record for the plone-legacy bundle names the only javascript resource left in Plone that does not work with the Resource registry. Note that any JavaScript or CSS registered with the old `portal_javascripts` or `portal_css` tools will be included automatically in this bundle. Note too that the plone-legacy bundle declares a dependency on the plone bundle, which ensures only that the plone bundle will be loaded into the page before this one.

```
<records prefix="plone.bundles/plone-legacy"
        interface='Products.CMFPlone.interfaces.IBundleRegistry'>
  <value key="resources" purge="false">
    <element>jquery-highlightsearchterms</element>
  </value>
  <value key="depends">plone</value>
  <value key="jscompilation">++plone++static/plone-legacy-compiled.js</value>
  <value key="csscompilation">++plone++static/plone-legacy-compiled.css</value>
  <value key="last_compilation">2014-08-14 00:00:00</value>
  <value key="compile">False</value>
  <value key="enabled">True</value>
</records>
```

A bundle is registered in the Plone add-on package `Plomino`. Here, a number of resources are aggregated and compiled via the `plone-compile-resources` script. They may also be compiled Through-The-Web, using the Resource Registry. Notice that in contrast to the plone bundle, the resources combined here all use `require` at the top level to avoid *The mismatched anonymous define error*. (see `Products/CMFPlomino/profiles/default/registry.xml` and for an example of the resources included `Products/CMFPlomino/browser/static/js/table.js`)

```
<records prefix="plone.bundles/plomino"
        interface='Products.CMFPlone.interfaces.IBundleRegistry'>
```

```

<value key="resources">
  <element>plominoformula</element>
  <element>plominotable</element>
  <element>plominodesign</element>
  <element>plominodynamic</element>
</value>
<value key="enabled">True</value>
<value key="depends">plone</value>
<value key="jscompilation">++resource++Products.CMFPlomino/js/plomino-compiled.js</
↪value>
<value key="csscompilation">++resource++Products.CMFPlomino/css/plomino-compiled.css
↪</value>
<value key="last_compilation">2015-12-08 00:00:00</value>
</records>

```

In **Rapido**, another Plone add-on, the JavaScript registered for the bundle is manually compiled. By listing the plone default bundle as a dependency, this JavaScript is able to rely on Plone default resources such as jQuery, mockup and the patterns registry being present. (see `rapido/plone/profiles/default/registry.xml` and `rapido/plone/browser/rapido.js`)

```

<records prefix="plone.bundles/rapido"
  interface='Products.CMFPlone.interfaces.IBundleRegistry'>
  <value key="enabled">True</value>
  <value key="jscompilation">++resource++rapido.js</value>
  <value key="csscompilation"></value>
  <value key="last_compilation">2019-11-26 00:00:00</value>
  <value key="compile">False</value>
  <value key="depends">plone</value>
</records>

```

Non-AMD Bundles

Sometimes it may be useful to register a simple javascript without using the AMD pattern. An example of such a bundle is provided in the `example.p4p5` package. In this case, there is a simple JavaScript which appends a status div to a chart (`example/p4p5/browser/static/chart.js`):

```

$(document).ready(function() {
  var chart = $('#chart');
  var done = parseInt(chart.attr('done'));
  var inprogress = parseInt(chart.attr('inprogress'));
  var total = done + inprogress;
  if(total == 0) {
    total = 1;
  }
  var done_rate = Math.round(100 * done / total);
  var inprogress_rate = Math.round(100 * inprogress / total);
  chart.append('<div class="done" style="width:'+done_rate+'%">&nbsp;</div>');
  chart.append('<div class="inprogress" style="width:'+inprogress_rate+'%">&nbsp;</
↪div>');
});

```

In this case, the JavaScript is dependent only on a global \$ which is expected to be bound to jQuery. Plone provides this in the plone bundle, so that is the only dependency we need to specify. For such a case, the package can register this JavaScript in `jsregistry.xml` for Plone versions before 5.0. And in Plone 5, the following bundle record added in `registry.xml` will do the trick (`example/p4p5/profiles/plone5/registry.xml`):

```
<records prefix="plone.bundles/examplep4p5"
  interface='Products.CMFPlone.interfaces.IBundleRegistry'>
  <value key="enabled">True</value>
  <value key="jscompilation">++resource++example.p4p5/chart.js</value>
  <value key="csscompilation">++resource++example.p4p5/chart.css</value>
  <value key="last_compilation">2016-01-01 00:00:00</value>
  <value key="compile">False</value>
  <value key="depends">plone</value>
</records>
```

Notice that this bundle provides *no resources*. The JavaScript file from the package is provided as the value of the `jscompilation` option. The CSS file is likewise provided as a pre-compiled value. Finally the value of the `compile` option is set to `False`. This ensures that the Resource Registry will make no attempt to re-compile this bundle.

Controlling Resource and Bundle Rendering

To control whether a bundle is included in a rendered page, we have already discussed several options. You may globally enable or disable a bundle using the `enabled` option of the bundle record. You may conditionally render the bundle using the `expression` option of the bundle record.

A Diazo Theme may also include or exclude specific bundles, regardless of whether they are enabled or disabled in the Resource Registry. To do so, use the `enabled-bundles` or `disabled-bundles` settings in the `manifest.cfg` file for the theme. These settings take a comma separated list of the names of bundles.

A browser page can include or exclude a specific bundle by using the API methods from `Products.CMFPlone.resources`. This will override the value of `enabled` in the Resource Registry for the named bundle.

Here are the API methods (from `Products.CMFPlone.resources`):

`add_bundle_on_request(request, bundle)`: The value provided for the `bundle` parameter must be the name of a bundle. The named bundle will be added to the provided request.

`remove_bundle_on_request(request, bundle)`: The value provided for the `bundle` parameter must be the name of a bundle. The named bundle will be removed from the provided request if it is present.

A browser page may also force the rendering of an individual resource on a particular request. Thus specific resources may be included regardless of whether they are included in a rendered bundle.

Here is the API method to do so (from `Products.CMFPlone.resources`):

`add_resource_on_request(request, resource)`: The value provided for `resource` must be the name of a resource. The named resource will be added to the current request.

Aggregate Bundles for Production

Plone defines several bundles. Add-ons that you include in your Plone site may also define bundles of their own. In production, *each* of these bundles will result in the loading of one JavaScript and one CSS file. To reduce the number of loaded files to an absolute minimum, we use “bundle aggregation”.

There are two bundle aggregations available in Plone. A first aggregation named `default` contains all the bundles that must be available at all times. It creates 2 output files (one JavaScript and one CSS). A second aggregation named `logged-in` contains bundles only needed for authenticated users. It also creates 2 output files (one JavaScript and one CSS).

Aggregation of bundles is triggered by the `registry.xml` Generic Setup import step. Installing any profile containing a `registry.xml` file will automatically refresh the current aggregations. Any bundles declared in that file will be included, if they declare that they should be merged with one of the two available aggregations.

As bundles can be defined or modified Through-The-Web, Plone also provides a “Merge bundles for production” button in the Resource Registry. This allows us to re-generate the aggregations manually after any Through-The-Web modifications have been made.

Declare an Aggregation

Custom bundles from an add-on or from a theme may be aggregated with the standard Plone bundles. To do so, use the `merge_with` option in your bundle declaration in `registry.xml`. The valid values are `default` or `logged-in`. If the `merge_with` option is not present or is empty, the bundle will not be aggregated and is published separately.

```
<records prefix="plone.bundles/my-bundle"
         interface='Products.CMFPlone.interfaces.IBundleRegistry'>
  <value key="merge_with">logged-in</value>
  ...
</records>
```

Note: Bundles cannot be conditionally included in an aggregation. If the value of the `merge_with` option is *default* or *logged-in*, the value of the `expression` option **will be ignored**.

Note: In Development mode, aggregation is disabled, all bundles are published separately.

Diazo Bundles

The point with Diazo is to create standalone static themes which work without Plone. Diazo themes can use - and will use - their own resources and compiling systems.

Diazo was extended to support bundles. Bundles can be defined in the theme’s `manifest.cfg` file.

Bundles configured in the `manifest.cfg` file are included in the output by the renderer additionally to the ones registered in the resource registry. This allows us to just overwrite or drop the `link` and `script` tags from the theme but still include the theme-specific resources without having to register them in the resource registry.

The options are:

enabled-bundles / disabled-bundles: A comma-separated list of Resource Registry bundles that should be included or excluded when rendering the Diazo theme. See [Controlling Resource and Bundle Rendering](#).

development-css / development-js: Uncompiled/unminified Less/CSS file and RequireJS files which should be included in development environments. Any required compilation will be handled by the browser on the fly.

production-css / production-js: Compiled CSS or JavaScript files that will be included in production mode.

tinymce-content-css: A CSS file to include for the TinyMCE editor. This allows theme developers to ensure that TinyMCE gives you the best possible WYSIWYG experience.

Note: Files referenced by `production-css` and `production-js` must be present in the theme and pre-compiled. Less and RequireJS files named in Diazo Bundles cannot be compiled by the Resource Registry Through-The-Web. Nor can they be compiled by the `plone-compile-resources` script. For Diazo Bundles, the theme must provide its own compilation toolchain.

Example manifest.cfg

This example is from `plonetheme.barceloneta`, the default theme in Plone 5 (`plonetheme/barceloneta/theme/manifest.cfg`). Here, a Less file for development, a compiled CSS file for production and a second compiled CSS file meant specifically for use with TinyMCE are all named. The `package` itself provides a `Gruntfile.js` and `package.json` file for compiling Less to CSS.

```
[theme]
title = Barceloneta Theme
description = The default theme for Plone 5
preview = preview.png
rules = /++theme++barceloneta/rules.xml
prefix = /++theme++barceloneta
doctype = <!DOCTYPE html>
enabled-bundles =
disabled-bundles =

development-css = /++theme++barceloneta/less/barceloneta.plone.less
production-css = /++theme++barceloneta/less/barceloneta-compiled.css
tinymce-content-css = /++theme++barceloneta/less/barceloneta-compiled.css

development-js =
production-js =
```

Migrating Older Add-ons

Many add-ons in the Plone ecosystem include JavaScript and CSS resources. To take advantage of the dependency management capabilities of the new Resource Registry, they will need to be migrated.

Compatibility With Deprecated Registries

The `portal_css` and `portal_javascript` registries have been deprecated in Plone 5. Older Add-ons register CSS and JavaScript resource with these registries using the `cssregistry.xml` and `jsregistry.xml` Generic Setup import steps. Plone 5 will still recognize these import steps, and resources registered with them will be added to the *plone-legacy bundle*.

Thus, older add-ons with JavaScript and CSS have a reasonable chance of working without migrating...yet.

However, scripts included in this fashion receive none of the dependency management benefits of the new Resource Registry. The *plone-legacy bundle* includes a global jQuery object and then includes bundled resources in order. The `define` and `require` APIs from RequireJS are unset before the *plone-legacy bundle* is included, and then re-defined after.

Updating non-AMD scripts

To take advantage of the dependency management of the new Resource Registry, you should upgrade your existing JavaScript files to use the AMD pattern. To do so, wrap existing JavaScript using this recipe:

```
require([
    'jquery',
    'other-library'
], function($, otherLibrary) {
    'use strict';
```



```
...  
// All the previous JavaScript file code here  
...  
});
```

(For a description of the `require(Array, Function)` used here, [See the AMD API documentation](#))

Dependencies required by your JavaScript code must be listed in the `Array` argument to the `require` API. You must use the RequireJS name identifiers of your dependencies here. These will be the names of the Plone Resources which provide those JavaScript modules.

Listed dependencies are be passed to the `Function` argument as parameters. They will be available to the code inside this function.

Register your modeified files as *resources* in `registry.xml`. Finally, register a *bundle* in `registry.xml` which includes any of your resources.

Note: When using `require` instead of `define`, the anonymous function is immediately called. If you would use `define` instead, you'd have to make a `require` call somewhere, with the dependency to your resource.

This recipe should work for many JavaScript files. Other patterns for module definition may be found in the [AMD API definitions](#) or the [RequireJS API documentation](#).

The mismatched anonymous define error

If you have worked with RequireJS before, you are likely to be aware of the *mismatched anonymous define()* error. It arises from misuse of the `require` and `define` APIs.

To work in RequireJS, code that uses a call to `define` must be loaded into a page **only** through a call to `require`. You may not load such code using a `<script>` tag.

When applied to the concept of resources and bundles this means that bundles should **only** ever be `require` calls. If you try to use a JavaScript file that has a `define` call with a bundle, you'll likely cause the *mismatched anonymous define()* error. Make sure to use a JavaScript file with a `require` call to include all your `define` resources.

This is a fact of how RequireJS works. It is normal behavior. Keeping it in mind can save you headaches.

Including non-RequireJS scripts in a Diazo theme

We have already described how to add resources to the legacy bundle. We have also discussed that the legacy bundle unsets the `define` and `require` statements before loading its resources so as to avoid the *mismatched anonymous define()* error and other possible problems.

If you have scripts in your Diazo theme that you don't want to register with the resource registry and which are not compatible with RequireJS, you can take a similar approach. Add these scripts below the Plone scripts and unset `define` and `require` yourself.

Here is an example Diazo rule which does so:

```
<before theme="/html/head/script[1]">                                <!-- ... before your own_<br>↪scripts -->                                                         <!-- include the Plone_<br>    <xsl:apply-templates select="/html/head/script" />               <!-- and then unset require_<br>↪scripts -->                                                         <br>    <script>                                                           <br>↪and define -->
```



```

        require = undefined
        define = undefined
    </script>
</before>

```

Barceloneta theme

Barceloneta is the name of the Plone 5 default theme. It's named after the [Barcelona beach and neighbourhood](#). Barceloneta is a Diazo theme made from scratch using modern frontend technologies. It's responsive and spans through all the Plone UI including the CMS backend part.

It's based on [Bootstrap 3](#), but it's not dependent of it in any way. Although it reuses some of the structure and good practices of the original Bootstrap, it has its own personality and is fully adapted to Plone.

Structure

Barceloneta uses [LESS](#) as a pre-processor to generate the resultant stylesheet. The LESS resources live in the `plonetheme.barceloneta` egg in the `plonetheme/barceloneta/theme/less` directory:

```

plonetheme/barceloneta/theme/less
- accessibility.plone.less
- alerts.plone.less
- barceloneta-compiled.css
- barceloneta-compiled.css.map
- barceloneta.css
- barceloneta.plone.export.less
- barceloneta.plone.less
- barceloneta.plone.local.less
- behaviors.plone.less
- breadcrumbs.plone.less
- buttons.plone.less
- code.plone.less
- contents.plone.less
- controlpanels.plone.less
- deco.plone.less
- discussion.plone.less
- dropzone.plone.less
- event.plone.less
- fonts.plone.less
- footer.plone.less
- forms.plone.less
- formtabbing.plone.less
- grid.plone.less
- header.plone.less
- image.plone.less
- loginform.plone.less
- main.plone.less
- mixin.borderradius.plone.less
- mixin.buttons.plone.less
- mixin.clearfix.plone.less
- mixin.forms.plone.less
- mixin.grid.plone.less
- mixin.gridframework.plone.less
- mixin.images.plone.less
- mixin.prefixes.plone.less

```

```
- mixin.tabfocus.plone.less
- modal.plone.less
- normalize.plone.less
- pagination.plone.less
- pickadate.plone.less
- plone-toolbarlogo.svg
- portlets.plone.less
- print.plone.less
- roboto
- scaffolding.plone.less
- search.plone.less
- sitemap.plone.less
- sitenav.plone.less
- sortable.plone.less
- states.plone.less
- tables.plone.less
- tablesorter.plone.less
- tags.plone.less
- thumbs.plone.less
- toc.plone.less
- tooltip.plone.less
- tree.plone.less
- type.plone.less
- variables.plone.less
- views.plone.less
```

They are divided by base styling, layout, function, components and views, so they could be reusable and extended from other themes. The main LESS resource that imports all the others is `barceloneta.plone.less`.

It has a set of LESS variables that can be overridden either through the web using the [Theming control panel](#) or by reusing it in your own theme. They include colors, sizes, fonts and other useful parameters.

Barceloneta makes use of the new [Diazo bundle](#) to expose its resources to Plone using the Resource Registries. As it is a pure Diazo theme, it keeps a low profile being Plone agnostic and only containing the theme itself.

Changes from previous versions of Plone

Regarding markup and comparing to the previous versions of Plone, Barceloneta introduced lots of changes in the default Plone markup to modernize it and make it more accessible. There are few parts of rendering Plone that were not updated.

However, any class or id that was stripped away from Plone was done with the purpose of making upgrades and adaptations of existing Diazo themes easy. Whenever possible additional classes and ids were introduced being always domain namespaced `plone-*`.

Register LESS resources profile

Barceloneta provides an optional GenericSetup profile that allows you to reuse the resources from the LESS files of your theme. This is done by registering all the Barceloneta LESS resources as Plone Resource Registries resources. This profile is called `plonetheme.barceloneta:registerless` and can be imported from an external theme GenericSetup profile `metadata.xml` like:

```
<?xml version="1.0"?>
<metadata>
  <version>1000</version>
  <dependencies>
```

```

<dependency>profile-plone.app.theming:default</dependency>
<dependency>profile-plonetheme.barceloneta:registerless</dependency>
</dependencies>
</metadata>

```

Using the barceloneta theme only for the backend

You can develop a custom Diazo based theme and use the Barceloneta theme only for the backend like follows shown below:

```

<?xml version="1.0" encoding="UTF-8"?>
<rules
  xmlns="http://namespaces.plone.org/diazo"
  xmlns:css="http://namespaces.plone.org/diazo/css"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xi="http://www.w3.org/2001/XInclude">

  <!-- Include the backend theme -->
  <xi:include href="++theme++barceloneta/backend.xml" />

  <!-- Only theme front end pages -->
  <rules css:if-content="body.frontend#visual-portal-wrapper">

    <theme href="index.html" />

    <!-- Your diazo front end rules go here -->

  </rules>
</rules>

```

You can define your own Diazo bundle (JavaScript and Less/CSS) in your manifest.cfg file by using the options `development-js`, `production-js`, `development-css` and `production-css`. This bundle will not be included in the backend theme.

Current issues

You will still need to include a minimal plone bundle in your theme for rendering the toolbar correctly. It is intended in future versions of Plone that this will be available by default and be very minimal making no assumptions about the JS or CSS of your frontend theme so as not to conflict with it.

Why this is a good idea

- It reduces the effort in theming.
- In most cases your users will never see edit, sharing, sitesetup or other aspects of the Plone backend UI.
- Making those screens work with a new theme is a lot of work.
- The backend pages can include a lot of add on functionality which might be hard to integrate.
- This might not be tested for integration into third-party themes.
- Barceloneta has been tested for UI and to some extend accessibility.
- Retheming could make the UI harder to use for editor.

- The backend UI is more likely to change between versions.
- Theming it means your theme will have to change too.

How this works

- There is a body class tag “frontend”.
- This appears when current view or page is unprotected or only protected by a “can view” permission.
- In most cases this your “view” of an object, and some extra pages like contact-us, login_form etc.
- Almost everything else is protected by other permissions and are therefore intended to be used by logged in users.
- `++theme++barceloneta/backend.xml` is mainly the same as the normal barceloneta rules except for a few exceptions:
 - It will only apply theming when `body.frontend` is not present
 - Except it will include the toolbar regardless if `body.frontend` is there or not.
 - It disables all popups. This makes it possible to switch theme using just the toolbar
 - It removes headers, footers and most “theme” elements from backend pages.

Inheriting a new theme from Barceloneta

Note: based on [Customize Plone 5 default theme on the fly](#) by Asko Soukka.

If you do not want to build a complete theme from scratch, you can use Barceloneta and just make small changes.

Create a new theme in the theming editor containing the following files:

- `manifest.cfg`, declaring your theme:

```
[theme]
title = mytheme
description =
development-css = /++theme++mytheme/styles.less
production-css = /++theme++mytheme/styles.css
```

- `rules.xml`, including the Barceloneta rules:

```
<?xml version="1.0" encoding="UTF-8"?>
<rules
  xmlns="http://namespaces.plone.org/diazo"
  xmlns:css="http://namespaces.plone.org/diazo/css"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xi="http://www.w3.org/2001/XInclude">

  <!-- Import Barceloneta rules -->
  <xi:include href="++theme++barceloneta/rules.xml" />

  <rules css:if-content="#visual-portal-wrapper">
    <!-- Placeholder for your own additional rules -->
  </rules>
```

```
</rules>
```

- a copy of `index.html` from Barceloneta (this one cannot be imported or inherited, it must be local to your theme).
- `styles.less`, importing Barceloneta styles:

```
/* Import Barceloneta styles */
@import "++theme++barceloneta/less/barceloneta.plone.less";

/* Customize whatever you want */
@plone-sitenav-bg: pink;
@plone-sitenav-link-hover-bg: darken(pink, 20%);
.plone-nav > li > a {
    color: @plone-text-color;
}
```

Then you have to compile `styles.less` to obtain your `styles.css` file using the “Build CSS” button.

Now your theme is ready. You can keep it in the theming editor, or you can export it and put the files in your theme add-on.

An older (Plone 4.2) quick guide which may help to understand Diazo better:

Quick Test Recipe

Description

Diazo is the system used to implement Plone themes. As of Plone 4.2, Plone ships with all the ‘machinery’ required to get started with Diazo based theme creation. This recipe is designed to get you started quickly.

The goal of this recipe is to help you confirm that everything is working. The theme resources for this recipe are hosted on a github page (<http://pigeonflight.github.io>).

Ingredients

You will need to have the following:

- Administrative access to a working copy of Plone 4.2 or (Plone 4.1 with `plone.app.theming` installed)

Procedure

If you’re using Plone 4.2, look for ‘Site Setup’ > ‘Theming’.

The screenshot shows the Plone 5 administration interface. At the top, there is a navigation bar with 'Home', 'News', 'Events', and 'Users' tabs. The 'Home' tab is selected. Below the navigation bar, there is a search bar and a dropdown menu for 'admin'. The main content area is titled 'Site Setup' and contains a warning message: 'Warning: You have not configured a mail host or a site 'From' address, various features including contact forms, email notification and password reset will not work. Go to the Mail control panel to fix this.' Below the warning, there is a section titled 'Plone Configuration' which lists various configuration options. The 'Theming' option is highlighted with a red circle.

Plone

admin

Search Site

only in current section

Home News Events Users

You are here: Home

Site Setup

Configuration area for Plone and add-on Products.

Warning You have not configured a mail host or a site 'From' address, various features including contact forms, email notification and password reset will not work. Go to the [Mail control panel](#) to fix this.

Plone Configuration

- Add-ons
- Calendar
- Configuration Registry
- Content Rules
- Discussion
- Editing
- Errors
- HTML Filtering
- Image Handling
- Language
- Mail
- Maintenance
- Markup
- Navigation
- Search
- Security
- Site
- Theming**
- TinyMCE Visual Editor
- Types
- Users and Groups
- Zope Management Interface

In a Plone 4.1 with `plone.app.theming` you may find the same thing under 'Site Setup' > 'Diazo theme' Instead.

admin ▼

Search Site Search

☐ only in current section

Home News Events Users Documents Images Files

Cool STuff

You are here: Home

Site Setup

Configuration area for Plone and add-on Products.

Warning You have not configured a mail host or a site 'From' address, various features including contact forms, email notification and password reset will not work. Go to the [Mail control panel](#) to fix this.

Plone Configuration

- Add-ons
- Calendar
- Collections
- Configuration Registry
- Content Rules
- Diazo theme**
- Discussion
- Editing
- Errors
- HTML Filtering
- Image Handling
- Language
- Mail
- Maintenance
- Markup
- Navigation
- Search
- Security
- Site
- Themes
- TinyMCE Visual Editor
- Types
- Users and Groups
- Zope Management Interface

Note: If you don't see anything like the 'Diazo theme' option, go to 'Site Setup' > 'Add-ons', select 'Diazo theme support' and click 'Activate'.

In the Diazo theming control panel click on the 'Advanced Settings' tab.

Theme settings

[Up to Site Setup](#)

Use this control panel to enable a Diazo theme. You can either select a pre-registered theme, or configure a theme directly by specifying a rules file.

Info Please note that this control panel page will never be themed.

Basic settings **Advanced settings** [Import theme](#)

Use the fields below to configure the Diazo rules file and absolute path prefix, or to enable reading a theme from a remote server.

Rules file
Enter a path or URL for the theme rules file.

Absolute path prefix
If your theme uses relative paths for images, stylesheets or other resources, you can enter a prefix here to make sure these resources will work regardless of which page Plone is rendering.

Doctype
You can specify a Doctype string which will be set on the output, for example "xhtml". If left blank the default XHTML 1.0 transitional Doctype or that set in the Diazo theme is used.

☒ **Read network**
Allow rules and themes to be read from remote servers.

Unthemed host names
If there are hostnames that you do not want to be themed, you can list them here, one per line. This is useful during theme development, so that you can compare the themed and unthemed sites. In some cases, you may also want to provide an unthemed site.

Enter the following values:

Rules file: `http://pigeonflight.github.io/diazodemo/rules.xml`

Absolute path prefix: `http://pigeonflight.github.io/diazodemo/`

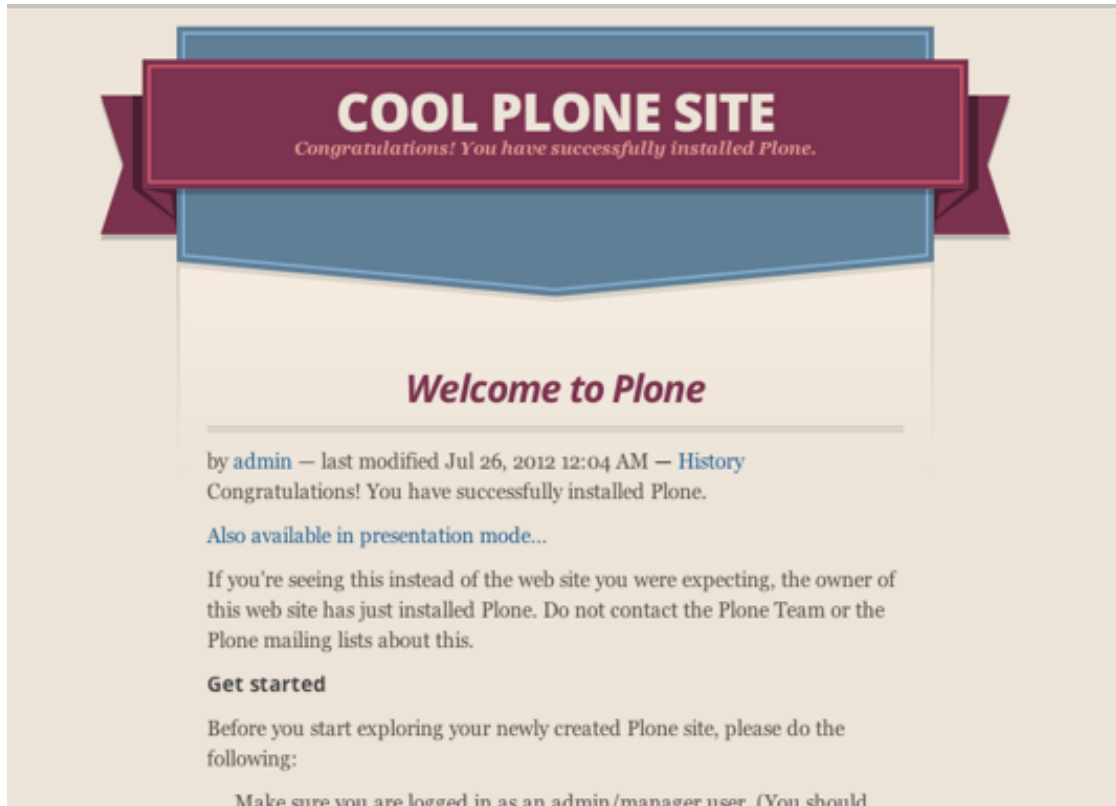
Read network should be checked, then click 'save'

Important: Make sure that your Diazo theme is enabled

Note: The rule file and resources in this example are hosted online, this will be a problem if your Plone site is behind a firewall or otherwise not connected to the internet.

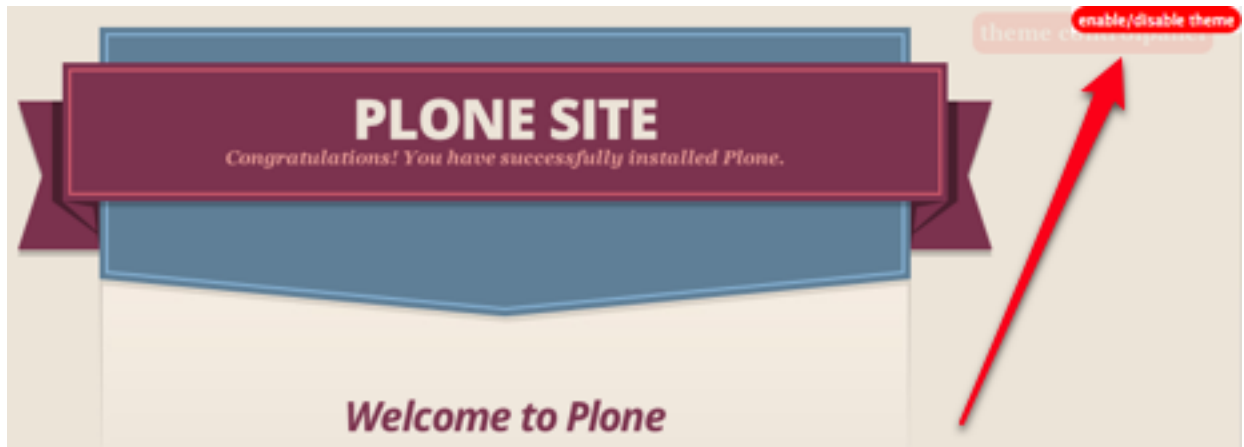
When you visit your Plone site you will see the main content displayed inside of the custom theme provided by <http://pigeonflight.github.io/diazodemo>.

It should look similar to this screenshot:



Disabling the test theme

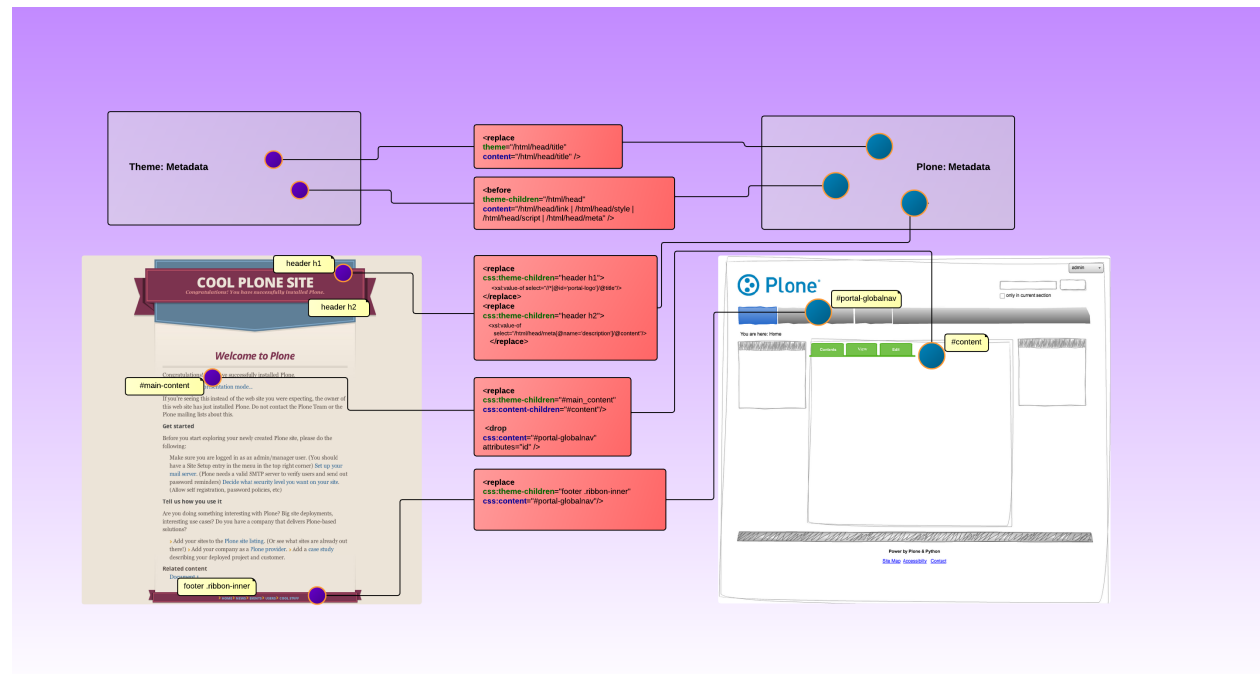
To disable the test theme click on the enable/disable button at the top right. Then uncheck the ‘Theme enabled’ box.



How the rule.xml file works

Think of the rules.xml file as a mapper which uses CSS ids and classes to identify content from the Plone site that should be injected into an HTML document.

The diagram below explains this visually.



View an explanatory diagram in PDF format

Troubleshooting

The theme is not showing

Check 'Site Setup' > 'Diazo Theme' and ensure that under 'Basic Settings', 'Enabled' is checked.

Using Diazo is also possible as a standalone service. Normally the Diazo theme transformation is running inside the Plone process. But you can compile the Diazo rules to low level XSLT and let a webserver do the actual transformation, or run the Diazo transformations in a [WSGI](#) enabled service. If you want this advanced stand alone set up, please take a look at documentation on www.diazo.org, especially the [Compilation](#) and [Deployment](#) chapters.

General information on the stylesheets and other resources in Plone

Front-end: templates, CSS and JavaScript

Instructions and information for front-end development for Plone CMS.

This includes creating page templates, managing JavaScript and CSS assets and writing JavaScript for Plone.

TAL page templates

Description

Plone uses Zope Page Templates (*ZPT*). This document contains references to this template language and generally available templates, macros and views you can use to build your Plone add-on product.

Introduction

Plone uses [Zope Page Templates](#), consisting of the three related standards: Template Attribute Language ([TAL](#)), TAL Expression Syntax ([TALES](#)), and Macro Expansion TAL ([METAL](#)).

A normal full Plone HTML page consists of:

- the *master template*, defining the overall layout of the page,
- *slots*, defined by the master template, and filled by the object being published,
- *viewlets and Viewlet managers*.

Templates can be associated with Python view classes (also known as “new style”, circa 2008) or they can be standalone (“old style”, circa 2001).

Note: The rationale for moving away from standalone page templates is that the page template code becomes cluttered with inline Python code. This makes templates hard to customize or override. New style templates provide better separation with view logic (Python code) and HTML generation (page template).

The MIME-Type

Basically a document file got a mime-type. This is also important for Plone Templates if you don’t want to export to text/html. If you want to export to a XML File you have to change the mime-type because otherwise the browser won’t recognize the file as an XML. At the moment Plone supports text/html which is the default value. And text/xml. You got 2 opportunities to change this value. If you customize a template you got an input box which called “Content-Type”. The other Way is to create a file named by your template name and extend the name by *.metadata*.

Example:

- my_view.pt
- my_view.pt.metadata

Content of metadata file:

```
[default]
content_type = text/xml
```

Overriding templates

The recommended approach to customize `.pt` files for Plone 4 is to use a little helper called [z3c.jbot](#).

If you need to override templates in core Plone or in an existing add-on, you can do the following:

- [Roll out your own add-on](#) which you can use to contain your page templates on the file system.
- Use the [z3c.jbot](#) Plone helper add-on to override existing page templates. This is provided in the [sane_plone_addon_template](#) add-in, no separate set-up needed.
- [z3c.jbot](#) can override page templates (`.pt` files) for views, viewlets, old style page templates and portlets. In fact it can override any `.pt` file in the Plone source tree.

Overriding a template using z3c.jbot

1. First of all, make sure that your customization add-on supports `z3c.jbot`. `sane_plone_addon_template` has a `templates` folder where you can drop in your new `.pt` files.
2. Locate the template you need to override in Plone source tree. You can do this by searching the `eggs/` folder of your Plone installation for `.pt` files. Usually this folder is `.../buildout-cache/eggs`.

Below is an example UNIX `find` command to find `.pt` files. You can also use Windows Explorer file search or similar tools:

```
$ find ~/code/buildout-cache/eggs -name "*.pt"
./archetypes.kss-1.4.3-py2.4.egg/archetypes/kss/browser/edit_field_wrapper.pt
./archetypes.kss-1.4.3-py2.4.egg/archetypes/kss/browser/view_field_wrapper.pt
./archetypes.kss-1.6.0-py2.6.egg/archetypes/kss/browser/edit_field_wrapper.pt
./archetypes.kss-1.6.0-py2.6.egg/archetypes/kss/browser/view_field_wrapper.pt
...
```

Note: Your `eggs/` folder may contain several versions of the same egg if you have re-run buildout or upgraded Plone. In this case the correct action is usually to pick the latest version.

3. Make a copy of `.pt` file you are going to override.

Rename the file to its so-called *canonical* name: to do this, exclude the `.egg` folder name from the filename, and then replace all slashes `/` with dot `.`:

```
archetypes/kss/browser/edit_field_wrapper.pt
```

to:

```
archetypes.kss.browser.edit_field_wrapper.pt
```

Drop the file in the `templates` folder you have registered for `z3c.jbot` in your add-on.

Make your changes in the new `.pt` file.

Warning: After overriding the template for the first time (adding the file to the `templates/` folder) you need to restart Plone. `z3c.jbot` scans new overrides only during the restart.

After the file is in place, changes to the file are instantly picked up: the template code is re-read on every HTTP request — just hit enter in your browser location bar. (Hitting enter in the location bar is quicker than hitting *Refresh*, which also reloads CSS and JS files.)

If you want to override an already overridden template, read here: <http://stackoverflow.com/questions/16209392/how-can-i-override-an-already-overriden-template-by-jbot>

More info:

- <https://pypi.python.org/pypi/z3c.jbot/>
- <http://blog.keul.it/2011/06/z3c.jbot-magical-with-your-skins.html>

Main template

The master page template in Plone is called `main_template.pt` and it is provided by the [Products.CMFPlone](#) package.

This template provides the visual frame for Plone themes. The template is an old-style page template living in `plone_skins/plone_templates`.

Custom per view main template

Here is an example how to provide a customized main template for one view. In this example we have customized main template so that only the content area is visible.

First we register our template in `configure.zcml`:

```
<!-- Provide a custom main_template for our consumption -->
<browser:page
    name="widgets-demo-main-template"
    for="*"
    permission="zope.Public"
    template="barebone-main-template.pt"
/>
```

We refer it in our page template instead of `here/main_template`:

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
      xmlns:tal="http://xml.zope.org/namespaces/tal"
      xmlns:metal="http://xml.zope.org/namespaces/metal"
      xmlns:i18n="http://xml.zope.org/namespaces/i18n"
      metal:use-macro="here/@@widgets-demo-main-template/macros/master"
      i18n:domain="plone.app.widgets"
      lang="en"
>
```

`barebone-main-template.pt` is an edited copy of `portal_skins/sunburst_templates/main_template.pt`:

```
<metal:page define-macro="master">
<tal:doctype tal:replace="structure string:&lt;!DOCTYPE html&gt;" />

<html xmlns="http://www.w3.org/1999/xhtml"
      tal:define="portal_state context/@@plone_portal_state;
                  context_state context/@@plone_context_state;
                  plone_view context/@@plone;
                  lang portal_state/language;
                  view nocall:view | nocall: plone_view;
                  dummy python: plone_view.mark_view(view);
                  portal_url portal_state/portal_url;
                  checkPermission nocall: context/portal_membership/checkPermission;
                  site_properties context/portal_properties/site_properties;
                  ajax_load request/ajax_load | nothing;
                  ajax_include_head request/ajax_include_head | nothing;
                  dummy python:request.RESPONSE.setHeader('X-UA-Compatible', 'IE=edge,chrome=1
↪ ');"
      tal:attributes="lang lang;">

<head>
```

```

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<metal:baseslot define-slot="base">
    <base tal:attributes="href plone_view/renderBase" /><!--[if lt IE 7]></base><!--
↪[endif]<!-->
</metal:baseslot>

<tal:notajax tal:condition="python:not ajax_load or ajax_include_head">
    <div tal:replace="structure provider:plone.htmlhead" />
    <link tal:replace="structure provider:plone.htmlhead.links" />

    <tal:comment replace="nothing">
        Various slots where you can insert elements in the header from a template.
    </tal:comment>
    <metal:topslot define-slot="top_slot" />
    <metal:headslot define-slot="head_slot" />
    <metal:styleslot define-slot="style_slot" />
    <metal:javasscriptslot define-slot="javascript_head_slot" />

    <meta name="viewport" content="width=device-width, initial-scale=0.6666, ↪
↪maximum-scale=1.0, minimum-scale=0.6666" />
    <meta name="generator" content="Plone - https://plone.org" />
</tal:notajax>
</head>

<body tal:define="isRTL portal_state/is_rtl;
    sl python:plone_view.have_portlets('plone.leftcolumn', view);
    sr python:plone_view.have_portlets('plone.rightcolumn', view);
    body_class python:plone_view.bodyClass(template, view);
    classes python:context.restrictedTraverse('@@sunburstview').
↪getColumnClasses(view)"
    tal:attributes="class body_class;
        dir python:isRTL and 'rtl' or 'ltr'">

<div id="visual-portal-wrapper">

    <div id="portal-columns" class="row">

        <div id="portal-column-content" class="cell" tal:attributes="class classes/
↪content">

            <div id="viewlet-above-content" tal:content="structure provider:plone.
↪abovecontent" tal:condition="not:ajax_load" />

            <metal:block define-slot="content">
                <div metal:define-macro="content"
                    tal:define="show_border context/@@plone/showToolbar; show_border ↪
↪python:show_border and not ajax_load"
                    tal:attributes="class python:show_border and 'documentEditable' ↪
↪or ''">

                    <div metal:use-macro="context/global_statusmessage/macros/portal_
↪message">

                        Status message
                    </div>

                    <metal:slot define-slot="body">
                        <div id="content">

```

```

        <metal:header define-slot="header" tal:content="nothing">
Visual Header
</metal:header>

        <metal:bodytext define-slot="main">

                <div id="viewlet-above-content-title" tal:content=
↪ "structure provider:plone.abovecontenttitle" tal:condition="not:ajax_load" />
                <metal:title define-slot="content-title">
                        <metal:comment tal:content="nothing">
                                If you write a custom title always use
                                <h1 class="documentFirstHeading"></h1> for it
                        </metal:comment>
                        <h1 metal:use-macro="context/kss_generic_macros/
↪ macros/generic_title_view">
                                Generic KSS Title. Is rendered with class=
↪ "documentFirstHeading".
                                </h1>
                        </metal:title>

                <div id="viewlet-below-content-title" tal:content=
↪ "structure provider:plone.belowcontenttitle" tal:condition="not:ajax_load" />

                <metal:description define-slot="content-description">
                        <metal:comment tal:content="nothing">
                                If you write a custom description always use
                                <div class="documentDescription"></div> for it
                        </metal:comment>
                        <div metal:use-macro="context/kss_generic_macros/
↪ macros/generic_description_view">
                                Generic KSS Description. Is rendered with class=
↪ "documentDescription".
                                </div>
                        </metal:description>

                <div id="viewlet-above-content-body" tal:content=
↪ "structure provider:plone.abovecontentbody" tal:condition="not:ajax_load" />

                <div id="content-core">
                        <metal:text define-slot="content-core" tal:content=
↪ "nothing">
                                Page body text
                        </metal:text>
                </div>

                <div id="viewlet-below-content-body" tal:content=
↪ "structure provider:plone.belowcontentbody" tal:condition="not:ajax_load" />

                </metal:bodytext>
        </div>
</metal:slot>

        <metal:sub define-slot="sub" tal:content="nothing">
                This slot is here for backwards compatibility only.
                Don't use it in your custom templates.
        </metal:sub>
</div>

```

```
        </metal:block>

    </div>
</div>
</body>
</html>

</metal:page>
```

Plone template element map

Plone 4 ships with the *Sunburst* theme. Its viewlets and viewlets managers are described [here](#).

Note: Plone 3 viewlets differ from Plone 4 viewlets.

Zope Page Templates

Zope Page Templates, or *ZPT* for short, is an XML-based templating language, consisting of the Template Attribute Language (*TAL*), TAL Expression Syntax (*TALES*), and Macro Expansion TAL (*METAL*).

It operates using two XML namespaces (`tal:` and `metal:`) that can occur either on attributes of elements in another namespace (e.g. you will often have *TAL* attributes on HTML elements) or on elements (in which case the element itself will be ignored, but all its attributes will be recognized as *TAL* or *METAL* statements).

A statement in the `tal:` namespace will modify the element on which it occurs and/or its child elements.

A statement in the `metal:` namespace defines how a template interacts with other templates (defining or using macros and slots to be filled by macros).

The value of an attribute in the `tal:` namespace is an expression. The syntax of this expression is defined by the *TALES* standard.

TAL

TAL is the Template Attribute Language used in Plone.

- [TAL Guide](#)

Escaped and unescaped content

By default, all *TAL* output is escaped for security reasons:

```
view.text = "<b>Test</b>"
```

```
<div tal:content="view/text" />
```

Will output escaped HTML source code:


```
&lt;t;b>Test</b>
```

Unescaped content can be output using the TALEs `structure` keyword in the expression for the `tal:replace` and `tal:content` statements:

```
<div tal:replace="structure view/text" />
```

Will output unescaped HTML source code:

```
<b>Test</b>
```

METAL

The *METAL* (Macro Expansion TAL) standard provides *macros* and *slots* to the template language.

Using METAL macros is no longer recommended, since they couple programming logic too tightly with the template language. You should use views instead.

Read more about them in the [TAL Guide](#).

TALES expressions

The value of TAL statements are defined by TALEs expressions. A TALEs expression starts with the expression type. If no type is specified, the default is assumed. Three types are standard:

- `path`: expressions (*default*),
- `python`: expressions,
- `string`: expressions.

They are generally useful, and not limited to use in Page Templates. For example, they are widely used in various other parts of Plone:

- CSS and JavaScript registries, to decide whether to include a particular file;
- Action conditions, to decide whether to show or hide action link;
- Workflow security guards, to decide whether to allow a workflow state transition
- etc.

Read more about expressions in [TAL Guide](#).

See the *Expressions chapter* for more information.

Omitting tags

Sometimes you need to create XML control structures which should not end up to the output page.

You can use `tal:omit-tag=""`:

```
<div tal:omit-tag="">
    Only the content of the tag is rendered, not the DIV tag itself.
</div>
```

Images

See *how to use images in templates*.

Overriding templates for existing Plone views

1. New style templates can be overridden by overriding the view using the template.
2. Old style templates can be overridden by register a new skins layer in `plone_skins`.

View page template

- <http://lionfacelemonface.wordpress.com/2009/03/02/i-used-macros-in-my-browser-views-and-saved-a-bunch-of-money-on-my->

Old style page template

- Create a new layer in `portal_skins`
- Templates are resolved by their name, and a property on the `portal_skins` tool defines the order in which skin layers are searched for the name (see the *Properties* tab on `portal_skins`).
- You can reorder layers for the active theme so that your layer takes priority.

Portlet slots

By default, Plone `main_template` has slots for left and right portlets. If you have a view where you don't explicitly want to render portlets you can do:

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
      xmlns:tal="http://xml.zope.org/namespaces/tal"
      xmlns:metal="http://xml.zope.org/namespaces/metal"
      xmlns:i18n="http://xml.zope.org/namespaces/i18n"
      lang="en"
      metal:use-macro="here/main_template/macros/master"
      i18n:domain="plone">

  <head>
    <metal:block fill-slot="column_one_slot" />
    <metal:block fill-slot="column_two_slot" />
  </head>
```

This blanks out the `column_one_slot` and `column_two_slot` slots.

Head slots

You can include per-template CSS and JavaScript in the `<head>` element using extra slots defined in Plone's `main_template.pt`.

Note that these media files do not participate in *portal_css* or *portal_javascript* resource compression.

Extra slots are:

```

<tal:comment replace="nothing"> A slot where you can insert elements in the header_
↳from a template </tal:comment>
<metal:headslot define-slot="head_slot" />

<tal:comment replace="nothing"> A slot where you can insert CSS in the header from a_
↳template </tal:comment>
<metal:styleslot define-slot="style_slot" />

<tal:comment replace="nothing"> This is deprecated, please use style_slot instead. </
↳tal:comment>
<metal:cssslot define-slot="css_slot" />

<tal:comment replace="nothing"> A slot where you can insert JavaScript in the header_
↳from a template </tal:comment>
<metal:javascriptslot define-slot="javascript_head_slot" />

```

Example use:

```

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
lang="en"
metal:use-macro="here/main_template/macros/master"
i18n:domain="sits">

  <metal:slot fill-slot="css_slot">
    <style media="all" type="text/css">

      .schema-browser {
        border-collapse: collapse;
      }

      .schema-browser td,
      .schema-browser th {
        vertical-align: top;
        border: 1px solid #aaa;
        padding: 0.5em;
        text-align: left;
      }

      .default {
        color: green;
      }

      .mandatory {
        color: red;
      }
    </style>
  </metal:slot>

  <body>
    <metal:main fill-slot="main">
      <p>
        Protocols marked with question marks can be required or not
        depending of the current state of the patient. For example,
        priodisability field depends on other set fields of the
        patient.
      </p>
      ...
    </metal:main>
  </body>
</html>

```

Edit frame

By default, Plone draws a green *edit* frame around the content if you can edit it. You might want to disable this behavior for particular views.

Hiding the edit frame

If you'd like to hide the (green) editing frame, place the following code in your Zope 2-style page template:

```
<metal:block fill-slot="top_slot"
    tal:define="dummy python:request.set('disable_border',1)" />
```

Examples of this usage:

- The [Contact info page](#).
- The [Recently modified page](#).

Special style on individual pages

To override page layout partially for individual pages you can use marker interfaces to register special overriding viewlets.

More information:

- [Viewlets](#)
- <http://starzel.de/blog/how-to-get-a-different-look-for-some-pages-of-a-plone-site>

URL quoting inside TAL templates

You need to escape TAL attribute URLs if they contain special characters like plus (+) in query parameters. Otherwise browsers will mangle links, leading to incorrect parameter passing.

Zope 2 provides `url_quote()` function which you can access

```
<td id="cal#"
    tal:define="std modules/Products.PythonScripts.standard;
                url_quote nocall: std/url_quote;
```

Then you can use this function in your TAL code

```
<a href="#" tal:define="start_esc python:url_quote(start)"
    tal:attributes="href string: ${url}/day?currentDate=${start_esc}&xmy=${xmy}&xsub=${
    ↪xsub}">
```

If you need to also quote spaces, use `url_quote_plus` rather than `url_quote`.

Using macros

Here is an example how to use `<metal:block define-macro="xxx">` and `<metal:block use-macro="xxx">` in your *view class* template files.

```

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:tal="http://xml.zope.org/namespaces/tal"
      xmlns:metal="http://xml.zope.org/namespaces/metal"
      xmlns:i18n="http://xml.zope.org/namespaces/i18n"
      tal:omit-tag=""
>

<metal:row define-macro="row">
  <!--
    A macro. You can call this using metal:use-macro
    and pass variables to using tal:define.
  -->
</metal:row>

<!-- Call macro in different parts of the main template using *widget* variable,
↳as a parameter -->

<table class="datagridwidget-table-view" tal:attributes="data-extra view/extra">

  <tbody class="datagridwidget-body">
    <tal:row repeat="widget view/getNormalRows">
      <tr>
        <metal:macro use-macro="template/macros/row" />
      </tr>
    </tal:row>

    <tal:row condition="view/getTTRow" define="widget view/getTTRow">
      <tr>
        <metal:macro use-macro="template/macros/row" />
      </tr>
    </tal:row>

    <tal:row condition="view/getAARow" define="widget view/getAARow">
      <tr>
        <metal:macro use-macro="template/macros/row" />
      </tr>
    </tal:row>

  </tbody>
</table>
</html>

```

More info

- <http://stackoverflow.com/q/13165748/315168>

CSS

Description

Creating and registering CSS files for Plone and Plone add-on products. CSS-related Python functionality.

Introduction

This page has Plone-specific CSS instructions.

In Plone, most CSS files are managed by the `portal_css` tool via the Management Interface. Page templates can still import CSS files directly, but `portal_css` does CSS file compression and merging automatically if used.

View all Plone HTML elements

To test Plone HTML element rendering go to `test_rendering` page on your site:

```
http://localhost:8080/Plone/test_rendering
```

It will output a styled list of all commonly used Plone user interface elements.

Registering a new CSS file

You can register stylesheets to be included in Plone's various CSS bundles using GenericSetup XML.

Example profiles/default/cssregistry.xml:

```
<?xml version="1.0"?>
<!-- Setup configuration for the portal_css tool. -->

<object name="portal_css">

  <!-- Stylesheets are registered with the portal_css tool here.
  You can also specify values for existing resources if you need to
  modify some of their properties.
  Stylesheet elements accept these parameters:
  - 'id' (required): it must respect the name of the CSS or DTML file
    (case sensitive). '.dtml' suffixes must be ignored.
  - 'expression' (optional, default: ''): a TAL condition.
  - 'media' (optional, default: ''): possible values: 'screen', 'print',
    'projection', 'handheld', ...
  - 'rel' (optional, default: 'stylesheet')
  - 'rendering' (optional, default: 'import'): 'import', 'link' or
    'inline'.
  - 'enabled' (optional, default: True): boolean
  - 'cookable' (optional, default: True): boolean (aka 'merging allowed')

  See registerStylesheet() arguments in
  ResourceRegistries/tools/CSSRegistry.py for the latest list of all
  available keys and default values.
  -->

  <stylesheet
    id="++resource++yourproduct.something/yourstylesheet.css"
    cacheable="True"
    compression="safe"
    cookable="True"
    enabled="1"
    expression=""
    media=""
    rel="stylesheet"
    rendering="import"
```

```
insert-after="ploneKss.css" />
</object>
```

In this case there should be a registered resource directory named `yourproduct.something`. In the directory should be a file `yourstylesheet.css`. If you have registered the stylesheet directly in `zcml`

```
<browser:resource name="yourstylesheet.css" file="yourstylesheet.css" />
```

then id must be

```
id="++resource++yourstylesheet.css"
```

Expressions

The `expression` attribute of `portal_css` defines when your CSS file is included on an HTML page. For more information see [expressions documentation](#).

Inserting CSS as last into anonymous bundles

Plone compresses and merges CSS files to *bundles*.

For Plone 3.x, the optimal place to put CSS file available to all users is after `ploneKss.css`, as in the example above, to override rules in earlier files.

CSS files for logged-in members only

Add the following expression to your CSS file:

```
not: portal/portal_membership/isAnonymousUser
```

If you want to load the CSS in the same bundle as Plone's default `member.css`, use `insert-after="member.css"`. In this case, however, the file will be one of the first CSS files to be loaded and cannot override values from other files unless the CSS directive `!important` is used.

Condition for Diazo themed sites

To check if theming is active, will return true if Diazo is enabled:

```
request/HTTP_X_THEME_ENABLED | nothing
```

Conditional comments (IE)

- <https://plone.org/products/plone/roadmap/232>

`cssregistry.xml` example:

```
<!-- Load stylesheet for IE6 - IE8 only to fix layout problems -->
<stylesheet
  id="++resource++plonetheme.xxx.stylesheets/ie.css"
  applyPrefix="False"
  authenticated="False"
```

```
cacheable="True"
compression="safe"
conditionalcomment="lt IE 9"
cookable="True"
enabled="1"
expression=""
media="screen"
rel="stylesheet"
rendering="link"
title=""
insert-before="ploneCustom.css" />
```

Generating CSS classes programmatically in templates

Try to put string generation code in your view/viewlet if you have one.

If you do not have a view (e.g. you're dealing with `main_template`) you can create a view and call it as in the following example.

View class generating CSS class spans:

```
from Products.Five.browser import BrowserView
from Products.CMFCore.utils import getToolByName

class CSSHelperView(BrowserView):
    """ Used by main_template <body> to set CSS classes """

    def __init__(self, context, request):
        self.context = context
        self.request = request

    def logged_in_class(self):
        """ Get CSS class telling whether the user is logged in or not

        This allows us to fine-tune layout when edit frame et. al.
        are on the screen.
        """
        mt = getToolByName(self.context, 'portal_membership')
        if mt.isAnonymousUser(): # the user has not logged in
            return "member-anonymous"
        else:
            return "member-logged-in"
```

Registering the view in ZCML:

```
<browser:view
    for="*"
    name="css_class_helper"
    class=".views.CSSHelperView"
    permission="zope.Public"
    allowed_attributes="logged_in_class"
/>
```

Calling the view in `main_template.pt`:

```
<body
    tal:define="css_class_helper nocall:here/@@css_class_helper"
```



```
tal:attributes="class string:${here/getSectionFromURL} template-${template/id} $
↪{css_class_helper/logged_in_class};
    dir python:test(isRTL, 'rtl', 'ltr')">
```

Defining CSS styles reaction to the presence of the class:

```
#region-content { padding: 0 0 0 0px !important;}
.member-logged-in #region-content { padding: 0 0 0 4px !important;}
```

Per-folder CSS theme overrides

- <https://pypi.python.org/pypi/Products.CustomOverrides>

Striping listing colors

In your template you can define classes for 1) the item itself 2) extra odd and even classes.

```
<div tal:attributes="class python:'feed-folder-item feed-folder-item-' + (repeat[
↪'child'].even() and 'even' or 'odd') ">
```

And you can colorize this with CSS:

```
.feed-folder-item {
    padding: 0.5em;
}

/* Make sure that all items have same amount of padding at the bottom,
whether they have last paragraph with margin or not.*/
#content .feed-folder-item p:last-child {
    margin-bottom: 0;
}

.feed-folder-item-odd {
    background: #ddd;
}

.feed-folder-item-even {
    background: white;
}
```

plone.css

plone.css is automagically generated dynamically based on the full portal_css registry configuration. It is used in e.g. TinyMCE to load all CSS styles into the TinyMCE <iframe> in a single pass. It is not used on the normal Plone pages.

plone.css generation:

- https://github.com/plone/Products.CMFPlone/blob/master/Products/CMFPlone/skins/plone_scripts/plone.css.py

CSS reset

If you are building a custom theme and you want to do a cross-browser CSS reset, the following snippet is recommended:

```
/* @group CSS Reset */

/* Remove implicit browser styles, to have a neutral starting point:
   - No elements should have implicit margin/padding
   - No underline by default on links (we add it explicitly in the body text)
   - When we want markers on lists, we will be explicit about it, and they render
   ↪ inline by default
   - Browsers are inconsistent about hX/pre/code, reset
   - Linked images should not have borders
   */

* { margin: 0; padding: 0; }
* :link, :visited { text-decoration: none; }
* ul, ol { list-style: none; }
* li { display: inline; }
* h1, h2, h3, h4, h5, h6, pre, code { font-size: 1em; }
* a img, :link img, :visited img { border: none; }
a { outline: none; }
table { border-spacing: 0; }
img { vertical-align: middle; }
```

Adding new CSS body classes

Plone themes provide certain standard CSS classes on the <body> element to identify view, template, site section, etc. for theming.

The default body CSS classes look like this:

```
<body class="template-subjectgroup portaltype-XXX-app-subjectgroup site-LS section-
↪courses icons-on" dir="ltr">
```

But you can include your own CSS classes as well. This can be done by overriding `plone.app.layout.globals.LayoutPolicy` class which is registered as the `plone_layout` view.

`layout.py`:

```
""" Override the default Plone layout utility.
"""

from zope.component import queryUtility
from zope.component import getMultiAdapter

from plone.i18n.normalizer.interfaces import IIDNormalizer
from plone.app.layout.globals import layout as base
from plone.app.layout.navigation.interfaces import INavigationRoot

class LayoutPolicy(base.LayoutPolicy):
    """
    Enhanced layout policy helper.

    Extend the Plone standard class to have some more <body> CSS classes
    """
```

```

based on the current context.
"""

def bodyClass(self, template, view):
    """Returns the CSS class to be used on the body tag.
    """

    # Get content parent
    body_class = base.LayoutPolicy.bodyClass(self, template, view)

    # Include context and parent ids as CSS classes on <body>
    normalizer = queryUtility(IIDNormalizer)

    body_class += " context-" + normalizer.normalize(self.context.getId())

    parent = self.context.aq_parent

    # Check that we have a valid parent
    if hasattr(parent, "getId"):
        body_class += " parent-" + normalizer.normalize(parent.getId())

    # Get path with "Default content item" wrapping applied
    context_helper = getMultiAdapter((self.context, self.request), name="plone_
↪context_state")
    canonical = context_helper.canonical_object()

    # Mark site front page with special CSS class
    if INavigationRoot.providedBy(canonical):

        if "template-document_view" in body_class:
            body_class += " front-page"

    # Add in logged-in / not logged in status
    portal_state = getMultiAdapter((self.context, self.request), name="plone_
↪portal_state")
    if portal_state.anonymous():
        body_class += " anonymous"
    else:
        body_class += " logged-in"

    return body_class

```

Related ZCML registration:

```

<browser:page
    name="plone_layout"
    for="*"
    permission="zope.Public"
    class=".layout.LayoutPolicy"
    allowed_interface="plone.app.layout.globals.interfaces.ILayoutPolicy"
/>

```

Resource folders

Description

How to use resource directories to expose static media files (css, js, other) in your Plone add-on product

Introduction

Resource folders are the Zope Toolkit way to expose static media files to Plone URL mapping.

Resource folders provide a mechanism which allows conflict free way to have static media files mapped to Plone URL space. Each URL is prefixed with ++resource++your.package resource identified.

ZCML resourceDirectory

If you want to customize media folder mapping point, you need to use the resourceDirectory directive.

Below is an example how to map *static* folder in your add-on root folder to be exposed via ++resource++your.product/URI

```
<configure
  xmlns="http://namespaces.zope.org/zope"
  xmlns:browser="http://namespaces.zope.org/browser">

  <!-- Resource directory for static media files -->
  <browser:resourceDirectory
    name="your.product"
    directory="static"
  />

</configure>
```

Skin layers

Description

Skin layers are a legacy Plone 2 technology, still is use, for adding overridable templates and media resources to Plone packages.

Introduction

Skin layers, portal_skins and CMFCore.SkinsTool are the old-fashioned way to manage Plone templates.

- Each Plone theme has set of folders it will pick from portal_skins. These sets are defined in portal_skins -> properties.
- Skins layers are searched for a template by template name, higher layers first.
- Skin layers can be reordered through-the-web in portal_skins -> properties

Defining a skin layer

Skin files are placed in the *skins* folder of your add-on product.

The structure looks like this:

- yourproduct/namespace/configure.zcml
- yourproduct/namespace/profiles/default/skins.xml
- yourproduct/namespace/skins
- yourproduct/namespace/skins/layer1folder
- yourproduct/namespace/skins/layer2folder/document_view.pt
- yourproduct/namespace/skins/layer2folder
- ...

GenericSetup skins.xml:

```
<?xml version="1.0"?>
<object name="portal_skins" meta_type="Plone Skins Tool">
  <object name="headeranimation" meta_type="Filesystem Directory View"
    directory="plone.app.headeranimation:skins/headeranimation"/>
  <skin-path name="*">
    <layer name="headeranimation" insert-after="custom"/>
  </skin-path>
</object>
```

ZCML to register the layer:

```
<configure
  ...
  xmlns:cmf="http://namespaces.zope.org/cmf">

  <cmf:registerDirectory name="skins" directory="skins" recursive="True" />

</configure>
```

See also

- <https://mail.zope.org/pipermail/zope-cmf/2007-February/025650.html>

Unit testing and portal_skins

If you test templates in your unit testing code you might need to call `PloneTestCase._refreshSkinData()`:

```
def afterSetUp(self):
    # Must be called to load our add-on skins folders
    # for unit testing
    self._refreshSkinData()
```

Activating the current skin layer from a debug/ipzope shell

The skin needs to be initialised before its files can be accessed e.g. via `restrictedTraverse`:

```
portal.setupCurrentSkin()
```

Rendering a skin layer template

Templates must be bound to a context object before rendering. Plone acquisition magic maps templates as acquired attributes of all contentish objects.

Example:

```
# Any page object
doc = portal.doc

# portal_skins/plone_content/document_view.pt template bound to document
doc.document_view

# Resulting HTML is rendered when template object is called
doc.document_view()
```

Testing templates

Below is some example code how templates behave.

Example:

```
(Pdb) doc
<ATDocument at /plone/doc>
(Pdb) template = doc.document_view
(Pdb) template
<FSPageTemplate at /plone/document_view used for /plone/doc>
(Pdb) template._filepath
'/home/moo/workspace2/plone.app.headeranimation/plone/app/headeranimation/skins/
↳headeranimation/document_view.pt'
```

Nested folder overrides (z3c.jbot)

z3c.jbot allows to override any portal_skins based file based on its file-system path + filename.

Example jbot ZCML slug (no layers, unconditional overrides)

```
<configure
  xmlns="http://namespaces.zope.org/zope"
  xmlns:five="http://namespaces.zope.org/five"
  xmlns:il8n="http://namespaces.zope.org/il8n"
  xmlns:browser="http://namespaces.zope.org/browser"
  >

  <browser:jbot directory="jbot" />
```

Then your add-on has folder structure (example):

```
yourcompany.app/yourcompany/app/jbot
yourcompany.app/yourcompany/app/jbot/Products.TinyMCE.skins.tinymce.plugins.table.js.
↳table.js
yourcompany.app/yourcompany/app/jbot/Products.TinyMCE.skins.tinymce.plugins.table.
↳html.pt
```

For layered example (theme layer, add-on layer), see

- https://github.com/miohtama/sane_plone_addon_template/blob/master/youraddon/configure.zcml#L41

More info

- <https://pypi.python.org/pypi/z3c.jbot>
- <http://stackoverflow.com/questions/6161802/nested-overrides-in-portal-skins-folder>

Poking portal_skins

`portal_skins` is a persistent tool in Plone site root providing functions to manage skin layers. Its code mostly lives in `Products.CMFCore.SkinsTool`.

Available skin layers are directly exposed as *traversable* attributes:

```
(Pdb) for i in dir(portal_skins): print i
ATContentTypes
ATReferenceBrowserWidget
CMFEditions
COPY
COPY__roles__
ChangeSet
DELETE
...
plone_3rdParty
plone_content
plone_deprecated
plone_ecmascript
plone_form_scripts
plone_forms
plone_images
plone_login
plone_portlets
plone_prefs
plone_scripts
plone_styles
plone_templates
```

`portal_skins.getSkinSelections()` will list available skins.

You can edit a specific skin layer:

```
skin = portal_skins.getSkinByName("Go Mobile Default Theme")
```

`portal_skins.selections` is a *PersistentDict* object holding *skin name -> comma separated layer list* mappings.

Dumping a portal_skins folder to the filesystem

`qPloneSkinDump` can build a filesystem dump from `portal_skins` but it only works on Plone 2. If you need this functionality you can try to use this script ripped off `qPloneSkinDump`: <https://gist.github.com/silviot/5402869>. It is a WorksForMe quality script; replace the variables and run it with:

```
bin/instance run export_skin_folder.py
```

DTML

DTML technology has been phased out ten years ago.

Do not use it.

Note: Up to version 4.1, Plone was using an older style of theming. Using that is not considered *best practice* anymore. See [older versions of these docs](#) if you need the information.

Site Setup

Introduction and overview

This screen, which is available for the roles `Site Admin` and `Manager`, is where you can configure most aspects of your website.

```
*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Open test browser
    Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
    Import library  Remote  ${PLONE_URL}/RobotRemote

    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Remote ZODB Setup  ${FIXTURE}

    ${language} =  Get environment variable  LANGUAGE  'en'
    Set default language  ${language}

    Enable autologin as  Manager
    ${user_id} =  Translate  user_id
```



```

... default=jane-doe
${user_fullname} = Translate user_fullname
... default=Jane Doe
Create user ${user_id} Member fullname=${user_fullname}
Set autologin username ${user_id}

Test Teardown
Run keyword if sys.argv[0].startswith('bin/robot')
... Remote ZODB TearDown ${FIXTURE}

Suite Teardown
Run keyword if not sys.argv[0].startswith('bin/robot')
... Teardown Plone Site
Run keyword if sys.argv[0].startswith('bin/robot')
... Close all browsers

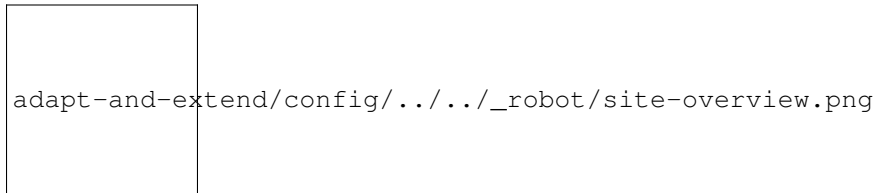
```

```

*** Test Cases ***

Show Site setup screen
Go to ${PLONE_URL}/@@overview-controlpanel
Capture and crop page screenshot
... ${CURDIR}/../../_robot/site-overview.png
... css=#content

```



When you have just created a new Plone site, you will see a warning here telling you to set up outgoing email for your site.

We'll come to that later. The configuration area is divided into several parts, leading to further setup screens.

Note: Many add-ons also come with their own setup area, you will also find these here after you have installed and activated them.

General

Date and Time This sets up the *date and time related settings like timezone*

Language *What languages are available.* Also how the URL scheme works for multilingual sites.

Mail Configure *outgoing email*. This is needed to register users, email password reset requests and the like.

Navigation Configure *how navigation is constructed*. Which content types should appear in navigation, should folders on the top level become tabs, and which workflows to show.

Site Various *site wide settings*: Site name, logo, metadata, settings for search engines and more.

Add-ons *Activate and deactivate* add-ons that enhance Plone's functionality.

Search *Various search setting*:: Activate live search, define which content types should be searched, etc.

Discussion *Comment settings*. Here you set whether you would like to permit commenting, if comments should be moderated and enable spam-protection mechanisms for comments.

Theming All about *how your site looks*. Enable your own theme, and work with the embedded theme editor.

Social Media Set *metadata for social media* such as Facebook app ID, twittercard ID.

Syndication Setting to *control RSS, Atom and itunes feeds* so your articles can be picked up by blog aggregators.

TinyMCE Settings for *the text editor*. Enable spell and grammar checking, add extra CSS classes for editors to use, etc.

Content

Content Rules Set up *automated mechanisms* to act on content when certain events occur. You can get an email when somebody adds a new item in a specific folder, and much more.

Editing Control *various editing settings*: which graphical editor to use, should automatic locking be performed when someone is editing, etc.

Image Handling Set up the *image sizes that Plone generates* and control image quality.

Markup Control if you want to *allow Markdown, Restructured Text* and other text formats.

Content Settings This is *where to control workflow, visibility and versioning of content*.

Dexterity Content types Here you can *create, adapt and extend* both the built-in content types, and your own ones. Define which fields are available, required, etc.

Users

Users and Groups *Create, define, delete and otherwise control* the users that can log in. Define groups and assign users to them, and define which properties (like email, address, or job position in your organisation) you would like to store.

Security

HTML Filtering Set *which kind of tags* you will allow users to enter. Malicious users, or users whose computer is infected by malware, can enter unwanted or dangerous content. Here you can finely choose what is acceptable and what now.

Security Various *security and privacy related settings*: Can users self-register? Should anonymous site visitors see author info on an article?

Error log This will *list errors and exceptions* that may have occurred recently. You can inspect them and store them, if wanted. These can point to potential problems with missing content, but also clues on security related matters.

Advanced

Note: The following can have large impact for your site. Take care when applying new settings.

Maintenance *Maintenance for the back-end database*. You can check the size of the database, and regularly *pack* it to keep it in optimal condition.

Management Interface This will take you to the *Management Interface*. In normal use, there is no need to go here. **Experts only**. But should you require access to the underlying software stack, here it is.

Caching Here you can *enable and fine-tune* Plone’s caching settings. This can have a great beneficial effect on the speed of your site, but make sure to read the documentation first.

Configuration Registry Provides *direct acces to all system variables*. Handle with care.

Resource Registries Provides *direct access to JavaScript, CSS and LESS resources*.

Date and Time

```
*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Open test browser
    Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
    Import library  Remote  ${PLONE_URL}/RobotRemote

    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Remote ZODB SetUp  ${FIXTURE}

    ${language} =  Get environment variable  LANGUAGE  'en'
    Set default language  ${language}

    Enable autologin as  Manager
    ${user_id} =  Translate  user_id
    ...  default=jane-doe
    ${user_fullname} =  Translate  user_fullname
    ...  default=Jane Doe
    Create user  ${user_id}  Member  fullname=${user_fullname}
    Set autologin username  ${user_id}

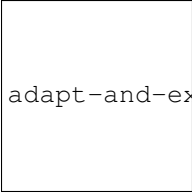
Test Teardown
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Remote ZODB TearDown  ${FIXTURE}

Suite Teardown
```

```
Run keyword if not sys.argv[0].startswith('bin/robot')
...           Teardown Plone Site
Run keyword if sys.argv[0].startswith('bin/robot')
...           Close all browsers
```

*** Test Cases ***

```
Show Date setup screen
Go to  ${PLONE_URL}/@@dateandtime-controlpanel
Capture and crop page screenshot
...   ${CURDIR}/../../_robot/date-setup.png
...   css=#content
```



adapt-and-extend/config/../../_robot/date-setup.png

You can set the default timezone, which is usually the one your server is in.

If you make more timezones available, users can set their own timezone. That way, dates and times will be converted to their time zone.

Note: This can be convenient, but potentially confusing. Imagine you are announcing a spectacular Event on your site: a gala fundraiser at the Ritz Theatre, Thursday, 21.00 hours. Logged-in visitors who are yet to travel from their timezone to your grand event will see the starting time in their own timezone.

Language settings

*** Settings ***

```
Resource plone/app/robotframework/server.robot
Resource plone/app/robotframework/keywords.robot
Resource Selenium2Screenshots/keywords.robot
```

```
Library OperatingSystem
```

```
Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown
```

*** Variables ***

```
${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content
```

*** Keywords ***

```
Suite Setup
    Run keyword if not sys.argv[0].startswith('bin/robot')
```

```

...          Setup Plone site  ${FIXTURE}
Run keyword if sys.argv[0].startswith('bin/robot')
...          Open test browser
Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
  Import library  Remote  ${PLONE_URL}/RobotRemote

  Run keyword if  sys.argv[0].startswith('bin/robot')
  ...            Remote ZODB SetUp  ${FIXTURE}

  ${language} = Get environment variable  LANGUAGE  'en'
  Set default language  ${language}

  Enable autologin as  Manager
  ${user_id} = Translate  user_id
  ...  default=jane-doe
  ${user_fullname} = Translate  user_fullname
  ...  default=Jane Doe
  Create user  ${user_id}  Member  fullname=${user_fullname}
  Set autologin username  ${user_id}

Test Teardown
  Run keyword if  sys.argv[0].startswith('bin/robot')
  ...            Remote ZODB TearDown  ${FIXTURE}

Suite Teardown
  Run keyword if  not sys.argv[0].startswith('bin/robot')
  ...            Teardown Plone Site
  Run keyword if  sys.argv[0].startswith('bin/robot')
  ...            Close all browsers

```

```

*** Test Cases ***

Show Language setup screen
  Go to  ${PLONE_URL}/@@language-controlpanel
  Capture and crop page screenshot
  ...  ${CURDIR}/../../../../_robot/language-setup.png
  ...  css=#content

  Click link  autotoc-item-autotoc-1
  Capture and crop page screenshot
  ...  ${CURDIR}/../../../../_robot/language-negotiation.png
  ...  css=#content

```



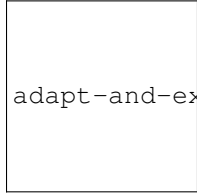
adapt-and-extend/config/../../../../_robot/language-setup.png

You can set up the default language for your site and the other languages it should be available in.

Note: This is for the language of the *user interface* of Plone. It will **not** make the content you create be translatable.

For that, you should enable the add-on “plone.app.multilingual” which is available in the [add-on section](#)

Should you have multiple languages, and plone.app.multilingual, enabled, you can set various options in the *Negotiation* tab:



adapt-and-extend/config/../../../../_robot/language-negotiation

Note: These are important choices, which are hard to change after you have made them, so think about them carefully when setting up multilingual sites.

The Plone community has much experience with multilingual sites, so do not hesitate to ask for advice on one of the support channels, see “Further help resources” in the footer of this website.

Mail Configuration

```
*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Open test browser
    Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
    Import library  Remote  ${PLONE_URL}/RobotRemote

    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Remote ZODB Setup  ${FIXTURE}
```

```

    ${language} = Get environment variable  LANGUAGE  'en'
    Set default language  ${language}

    Enable autologin as  Manager
    ${user_id} = Translate  user_id
    ...  default=jane-doe
    ${user_fullname} = Translate  user_fullname
    ...  default=Jane Doe
    Create user  ${user_id}  Member  fullname=${user_fullname}
    Set autologin username  ${user_id}

Test Teardown
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...  Remote ZODB TearDown  ${FIXTURE}

Suite Teardown
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...  Teardown Plone Site
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...  Close all browsers

```

```

*** Test Cases ***

Show Mail setup screen
    Go to  ${PLONE_URL}/@@mail-controlpanel
    Capture and crop page screenshot
    ...  ${CURDIR}/../../_robot/mail-setup.png
    ...  css=#content

```

adapt-and-extend/config/../../_robot/mail-setup.png

Completing this configuration allows your Plone site to send email. If the mail settings are not configured properly, you will not be able to receive form submissions via email from your site, and users can't be contacted for an email reset link.

Using localhost for email

One common way to configure mail for your Plone site is to use a mail server on the same machine that is hosting Plone. To do this, you'll first need to configure a mail server, like [Postfix](#).

SMTP Server: localhost

SMTP Port: 25

ESMTP Username: Leave this blank

ESMTP Password: Leave this blank

Site 'From' Name: [This will appear as the "From" address name]

Site 'From' Address: [emailaddress]@[yourdomain]

Using an external host

The following settings are an example of how you can configure your site to use your Gmail address. You can also use any external mail server, such as your business or institution email (you can get your SMTP settings from your in-house IT department).

SMTP Server: smtp.gmail.com

SMTP Port: 587

ESMTP Username: [username]@gmail.com

ESMTP Password: [Your Gmail Password]

Site ‘From’ Name: [This will appear as the “From” address name]

Site ‘From’ Address: [Your Gmail Address]

Testing the Configuration

You can test the configuration by clicking the “Save and send test e-mail” button at the bottom of the form. You should receive an email from the email address you specified with the subject “Test email from Plone.”

Navigation settings

```
*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...            Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...            Open test browser
    Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
    Import library  Remote  ${PLONE_URL}/RobotRemote

    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...            Remote ZODB Setup  ${FIXTURE}
```



```

    ${language} = Get environment variable  LANGUAGE  'en'
    Set default language  ${language}

    Enable autologin as  Manager
    ${user_id} = Translate  user_id
    ...  default=jane-doe
    ${user_fullname} = Translate  user_fullname
    ...  default=Jane Doe
    Create user  ${user_id}  Member  fullname=${user_fullname}
    Set autologin username  ${user_id}

Test Teardown
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...  Remote ZODB TearDown  ${FIXTURE}

Suite Teardown
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...  Teardown Plone Site
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...  Close all browsers

```

```
*** Test Cases ***
```

```

Show Navigation setup screen
    Go to  ${PLONE_URL}/@@navigation-controlpanel
    Capture and crop page screenshot
    ...  ${CURDIR}/../../_robot/navigation-setup.png
    ...  css=#content

```

adapt-and-extend/config/../../_robot/navigation-setup.png

Plone automatically generates navigation tabs from items you put in the *root*, or top level, of your site. The first two settings allow you to control this:

- do you want them to be generated?
- and if so, should it only be done for Folders or also other items like Pages?

The rest of the options control what items should be visible in the navigation. The defaults are usually a good choice, although many site administrators turn off “Image”. You can always re-visit this section later.

Site Configuration

```
*** Settings ***
```

```

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

```

```
Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Open test browser
    Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
    Import library  Remote  ${PLONE_URL}/RobotRemote

    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Remote ZODB SetUp  ${FIXTURE}

    ${language} =  Get environment variable  LANGUAGE  'en'
    Set default language  ${language}

    Enable autologin as  Manager
    ${user_id} =  Translate  user_id
    ...  default=jane-doe
    ${user_fullname} =  Translate  user_fullname
    ...  default=Jane Doe
    Create user  ${user_id}  Member  fullname=${user_fullname}
    Set autologin username  ${user_id}

Test Teardown
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Remote ZODB TearDown  ${FIXTURE}

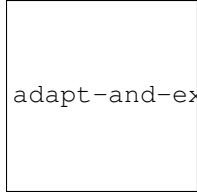
Suite Teardown
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Teardown Plone Site
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Close all browsers
```

```
*** Test Cases ***

Show Site setup screen
    Go to  ${PLONE_URL}/@@site-controlpanel
    Capture and crop page screenshot
    ...  ${CURDIR}/../../_robot/site-setup.png
    ...  css=#content
```

These settings should be changed for every site.

Site title The name of your website



adapt-and-extend/config/../../../../_robot/site-setup.png

Site Logo Upload your site logo. For extensive customization, you will want to create a special theme that will include your logo, but for a quick change this is enough.

Expose Dublin Core metadata This option allows information per content item, like Description, Tags, Author and others, to be shown to and ranked by search engines. It can help improve your search ranking, provided you fill in those fields with correct information.

Expose sitemap.xml.gz Almost always a good idea on a public website. It will make life for search engines easier, meaning they can better index your content.

JavaScript for web statistics support To gather information for web analytics like Piwik (an open-source self-hosted option) or Google Analytics you can paste the required snippet of code here. Be aware that this can have legal implications (so-called “cookie laws”) in some countries.

Display publication date Show the date a content item was published in the byline.

Icon visibility Controls whether to show different icons for different types of content. Can be useful for content editors, but distracting for anonymous visitors. You can set it to show only for logged-in users.

Toolbar position On modern wide-screen monitors, having the Toolbar to the side provides most usable space. But some people prefer it to be on the top of their screen.

Site based relative URL for toolbar Logo Customize the small logo on top of the toolbar, if you prefer. Note: this is not the same as the website logo.

robots.txt By convention, search engines look for a file called robots.txt to show them what they should index or not.

Add-ons

```
*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
```

```
...          Setup Plone site  ${FIXTURE}
Run keyword if sys.argv[0].startswith('bin/robot')
...          Open test browser
Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
  Import library  Remote  ${PLONE_URL}/RobotRemote

  Run keyword if  sys.argv[0].startswith('bin/robot')
  ...            Remote ZODB SetUp  ${FIXTURE}

  ${language} = Get environment variable  LANGUAGE  'en'
  Set default language  ${language}

  Enable autologin as  Manager
  ${user_id} = Translate  user_id
  ...  default=jane-doe
  ${user_fullname} = Translate  user_fullname
  ...  default=Jane Doe
  Create user  ${user_id}  Member  fullname=${user_fullname}
  Set autologin username  ${user_id}

Test Teardown
  Run keyword if  sys.argv[0].startswith('bin/robot')
  ...            Remote ZODB TearDown  ${FIXTURE}

Suite Teardown
  Run keyword if  not sys.argv[0].startswith('bin/robot')
  ...            Teardown Plone Site
  Run keyword if  sys.argv[0].startswith('bin/robot')
  ...            Close all browsers
```

```
*** Test Cases ***

Show Date setup screen
  Go to  ${PLONE_URL}/prefs_install_products_form
  Capture and crop page screenshot
  ...  ${CURDIR}/../../_robot/addon-setup.png
  ...  css=#content
```



adapt-and-extend/config/../../_robot/addon-setup.png

Here you can activate and deactivate add-ons.

Note: Before an add-on will show up here, you will first have to *install* it. See the [section on installing add-ons](#)

If you have *upgraded* add-ons, there are often *upgrade steps* to bring all configuration of your add-on to the new standard. This is also where you will be doing that.

Search settings

```

*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Open test browser
    Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
    Import library  Remote  ${PLONE_URL}/RobotRemote

    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Remote ZODB SetUp  ${FIXTURE}

    ${language} =  Get environment variable  LANGUAGE  'en'
    Set default language  ${language}

    Enable autologin as  Manager
    ${user_id} =  Translate  user_id
    ...  default=jane-doe
    ${user_fullname} =  Translate  user_fullname
    ...  default=Jane Doe
    Create user  ${user_id}  Member  fullname=${user_fullname}
    Set autologin username  ${user_id}

Test Teardown
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Remote ZODB TearDown  ${FIXTURE}

Suite Teardown
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Teardown Plone Site
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Close all browsers

```

```

*** Test Cases ***

```

```
Show Date setup screen
Go to  ${PLONE_URL}/@@search-controlpanel
Capture and crop page screenshot
...   ${CURDIR}/../../../../_robot/search-setup.png
...   css=#content
```



adapt-and-extend/config/../../../../_robot/search-setup.png

The “**Enable LiveSearch**” checkbox will start showing visitors the first results as they are typing in the search box.

Note: While this feature is very useful for smaller to medium-sized sites, it can be a performance hazard if you get hundreds of thousands or millions of visitors per day. If your site is very heavily visited, consider turning this off, and measuring the effect on server load.

For enterprise-sized sites, there are ready-made add-ons that will integrate with search solutions like Solr or Elastic-Search.

The other options allow you to specify which content types are to show up in search results, and when to crop the description.

More content types will appear here when you install or create new ones.

Discussion

```
*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Open test browser
```

```

Run keyword and ignore error Set window size @{{DIMENSIONS}}

Test Setup
  Import library Remote ${PLONE_URL}/RobotRemote

  Run keyword if sys.argv[0].startswith('bin/robot')
  ... Remote ZODB SetUp ${FIXTURE}

  ${language} = Get environment variable LANGUAGE 'en'
  Set default language ${language}

  Enable autologin as Manager
  ${user_id} = Translate user_id
  ... default=jane-doe
  ${user_fullname} = Translate user_fullname
  ... default=Jane Doe
  Create user ${user_id} Member fullname=${user_fullname}
  Set autologin username ${user_id}

Test Teardown
  Run keyword if sys.argv[0].startswith('bin/robot')
  ... Remote ZODB TearDown ${FIXTURE}

Suite Teardown
  Run keyword if not sys.argv[0].startswith('bin/robot')
  ... Teardown Plone Site
  Run keyword if sys.argv[0].startswith('bin/robot')
  ... Close all browsers

```

```

*** Test Cases ***

Show Discussion setup screen
  Go to ${PLONE_URL}/@@discussion-controlpanel
  Capture and crop page screenshot
  ... ${CURDIR}/../../../../_robot/discussion-setup.png
  ... css=#content

```

adapt-and-extend/config/../../../../_robot/discussion-setup.png

You can control all aspects of Plone's built-in Discussion module:

- do you want to enable comments at all?
- if yes, do you want to allow anonymous comments?
- do you want to moderate comments?

and various settings to protect your site from spam and malicious content.

For public sites, there are also add-ons available to integrate other comment solutions like Disqus, which have more robust spam protection.

But since these integrate external services, that is usually not acceptable on intranets. Then again, on an intranet, you will have far less trouble with anonymous and spammy commenting, and you may even trust the users there enough

to allow posting links.

Theming

Note: Please note that this control panel page will never be themed. This is a safety measure, so that even when a Theme is not working correctly you can get back to this screen to disable or edit it.

Note: The “Advanced settings” tab always configures the current active theme. If you change the theme, previously changed settings from another theme are lost.

```
*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Open test browser
    Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
    Import library  Remote  ${PLONE_URL}/RobotRemote

    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Remote ZODB Setup  ${FIXTURE}

    ${language} =  Get environment variable  LANGUAGE  'en'
    Set default language  ${language}

    Enable autologin as  Manager
    ${user_id} =  Translate  user_id
    ...  default=jane-doe
    ${user_fullname} =  Translate  user_fullname
    ...  default=Jane Doe
    Create user  ${user_id}  Member  fullname=${user_fullname}
    Set autologin username  ${user_id}
```



```

Test Teardown
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...            Remote ZODB TearDown  ${FIXTURE}

Suite Teardown
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...            Teardown Plone Site
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...            Close all browsers

```

```

*** Test Cases ***

Show Theme setup screen
    Go to  ${PLONE_URL}/@@theming-controlpanel
    Capture and crop page screenshot
    ...   ${CURDIR}/../../_robot/theme-setup.png
    ...   css=#content

```



adapt-and-extend/config/../../_robot/theme-setup.png

Fig. 4.1: For a full description on Diazo theming and the Theme editor, see [this section](#)

Social Media metadata

```

*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...            Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')

```

```
...          Open test browser
Run keyword and ignore error Set window size  @{DIMENSIONS}

Test Setup
  Import library  Remote  ${PLONE_URL}/RobotRemote

  Run keyword if  sys.argv[0].startswith('bin/robot')
  ...            Remote ZODB SetUp  ${FIXTURE}

  ${language} = Get environment variable  LANGUAGE  'en'
  Set default language  ${language}

  Enable autologin as  Manager
  ${user_id} = Translate  user_id
  ... default=jane-doe
  ${user_fullname} = Translate  user_fullname
  ... default=Jane Doe
  Create user  ${user_id}  Member  fullname=${user_fullname}
  Set autologin username  ${user_id}

Test Teardown
  Run keyword if  sys.argv[0].startswith('bin/robot')
  ...            Remote ZODB TearDown  ${FIXTURE}

Suite Teardown
  Run keyword if  not sys.argv[0].startswith('bin/robot')
  ...            Teardown Plone Site
  Run keyword if  sys.argv[0].startswith('bin/robot')
  ...            Close all browsers
```

```
*** Test Cases ***

Show socialmedia setup screen
  Go to  ${PLONE_URL}/@@social-controlpanel
  Capture and crop page screenshot
  ...  ${CURDIR}/../../_robot/social-setup.png
  ...  css=#content
```



adapt-and-extend/config/../../_robot/social-setup.png

Enabling this setting will add social media meta tags (Facebook OpenGraph and Twitter) to pages, so that when you share a Plone webpage on services like Facebook and Twitter, links and images will be better formatted.

Note: This **does not** add any “sharing buttons”, nor will it connect to the social media networks (often unacceptable in corporate or intranet settings). There is a variety of add-ons that provide this functionality.

Syndication settings

```

*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Open test browser
    Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
    Import library  Remote  ${PLONE_URL}/RobotRemote

    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Remote ZODB SetUp  ${FIXTURE}

    ${language} =  Get environment variable  LANGUAGE  'en'
    Set default language  ${language}

    Enable autologin as  Manager
    ${user_id} =  Translate  user_id
    ...  default=jane-doe
    ${user_fullname} =  Translate  user_fullname
    ...  default=Jane Doe
    Create user  ${user_id}  Member  fullname=${user_fullname}
    Set autologin username  ${user_id}

Test Teardown
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Remote ZODB TearDown  ${FIXTURE}

Suite Teardown
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Teardown Plone Site
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Close all browsers

```

```

*** Test Cases ***

```

```
Show Syndication setup screen
Go to  ${PLONE_URL}/@@syndication-controlpanel
Capture and crop page screenshot
...   ${CURDIR}/../../../../_robot/syndication-setup.png
...   css=#content
```



adapt-and-extend/config/../../../../_robot/syndication-setup.png

These settings will allow you to enable syndication of your content via various standard protocols.

By default, the well-supported RSS (versions 1 and 2), Atom and iTunes formats are supported.

You can enable these settings side-wide, or just for a specific Folder or Collection, for instance one that you create with the latest News items, press releases, or blog posts.

TinyMCE configuration

```
*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...             Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...             Open test browser
    Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
    Import library  Remote  ${PLONE_URL}/RobotRemote

    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...             Remote ZODB SetUp  ${FIXTURE}

    ${language} =  Get environment variable  LANGUAGE  'en'
```

```

Set default language  ${language}

Enable autologin as  Manager
${user_id} = Translate user_id
... default=jane-doe
${user_fullname} = Translate user_fullname
... default=Jane Doe
Create user  ${user_id}  Member  fullname=${user_fullname}
Set autologin username  ${user_id}

Test Teardown
Run keyword if  sys.argv[0].startswith('bin/robot')
...           Remote ZODB TearDown  ${FIXTURE}

Suite Teardown
Run keyword if  not sys.argv[0].startswith('bin/robot')
...           Teardown Plone Site
Run keyword if  sys.argv[0].startswith('bin/robot')
...           Close all browsers

```

```

*** Test Cases ***

Show Mail setup screen
Go to  ${PLONE_URL}/@@tinymce-controlpanel
Capture and crop page screenshot
...  ${CURDIR}/../../_robot/tinymce-setup.png
...  css=#content

```

adapt-and-extend/config/../../_robot/tinymce-setup.png

Here you can finetune the appearance and settings of TinyMCE, the default text editor.

On the “Plugins and Toolbar” screen, you can enable and disable buttons on TinyMCE’s toolbar.

The “Spell Checker” section holds a special bonus: you can enable the “After The Deadline” spell checker, which will highlight not only spelling mistakes but also grammar errors and common writing style errors.

Note: If you use the “After the Deadline” spellchecker in a security-conscious setting, or with many users, you are encouraged to set up your own instance of the server. The software is open-source, and not difficult to set up.

Templating in TinyMCE

TinyMCE in Plone 5 is adapted to allow templating engine for its content.

Right now parametrized templates are not implemented.

How to enable it

- On Control Panel -> TinyMCE -> Toolbar -> custom plugins add:

```
template|+plone+static/components/tinymce-built/js/tinymce/plugins/template
```

- On Control Panel -> TinyMCE -> Toolbar -> toolbar:

```
undo redo | styleselect | bold italic | alignleft aligncenter alignright |  
↪ alignjustify | bullist numlist outdent indent | unlink plonelink ploneimage |  
↪ template
```

How to configure which templates are available

For each template we need a file available on the browser, we assume for this example to use a diazo file at `++theme+example/templates/template.html` with the content:

```
<div class="mceTpl">  
  <h1>Template</h1>  
    
  <div class="row">  
    <div class="col-md-6">  
      <h2>Header</h2>  
    </div>  
    <div class="col-md-6">  
      <h2>Header</h2>  
    </div>  
  </div>  
</div>
```

In order to define it:

```
[  
  {  
    "title": "Template example",  
    "url": "++theme+example/templates/template.html",  
    "description": "Title with two columns"  
  },  
  {  
    "title": "Template example2",  
    "url": "++theme+example/templates/template2.html",  
    "description": "Title with three columns."  
  }  
<div class="mceTpl">  
  <h1>Template</h1>  
    
  <div class="row">  
    <div class="col-md-6">  
      <h2>Header</h2>  
    </div>  
    <div class="col-md-6">  
      <h2>Header</h2>  
    </div>  
  </div>  
</div>
```

Content Rules

Please see the [section on Content Rules](#) for full instructions on how to enable this powerful mechanism to automate dealing with content.

Editing

```
*** Settings ***  
  
Resource  plone/app/robotframework/server.robot  
Resource  plone/app/robotframework/keywords.robot  
Resource  Selenium2Screenshots/keywords.robot
```

```

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...            Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...            Open test browser
    Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
    Import library  Remote  ${PLONE_URL}/RobotRemote

    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...            Remote ZODB SetUp  ${FIXTURE}

    ${language} =  Get environment variable  LANGUAGE  'en'
    Set default language  ${language}

    Enable autologin as  Manager
    ${user_id} =  Translate  user_id
    ...  default=jane-doe
    ${user_fullname} =  Translate  user_fullname
    ...  default=Jane Doe
    Create user  ${user_id}  Member  fullname=${user_fullname}
    Set autologin username  ${user_id}

Test Teardown
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...            Remote ZODB TearDown  ${FIXTURE}

Suite Teardown
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...            Teardown Plone Site
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...            Close all browsers

```

```

*** Test Cases ***

Show Editing setup screen
    Go to  ${PLONE_URL}/@@editing-controlpanel
    Capture and crop page screenshot
    ...  ${CURDIR}/../../../../_robot/editing-setup.png
    ...  css=#content

```

“Link Integrity Checks” means that users will be warned if they try to delete an item that is being linked to from



adapt-and-extend/config/../../../../_robot/editing-setup.png

another content item. This setting is usually best left “on”

Likewise, it is best to leave the “Enable locking” setting on.

Image handling

```
*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Open test browser
    Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
    Import library  Remote  ${PLONE_URL}/RobotRemote

    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Remote ZODB SetUp  ${FIXTURE}

    ${language} =  Get environment variable  LANGUAGE  'en'
    Set default language  ${language}

    Enable autologin as  Manager
    ${user_id} =  Translate  user_id
    ...  default=jane-doe
    ${user_fullname} =  Translate  user_fullname
    ...  default=Jane Doe
    Create user  ${user_id}  Member  fullname=${user_fullname}
    Set autologin username  ${user_id}
```



```

Test Teardown
    Run keyword if sys.argv[0].startswith('bin/robot')
    ...           Remote ZODB TearDown  ${FIXTURE}

Suite Teardown
    Run keyword if not sys.argv[0].startswith('bin/robot')
    ...           Teardown Plone Site
    Run keyword if sys.argv[0].startswith('bin/robot')
    ...           Close all browsers

```

```

*** Test Cases ***

Show Image handling setup screen
    Go to  ${PLONE_URL}/@@imaging-controlpanel
    Capture and crop page screenshot
    ...   ${CURDIR}/../../_robot/imaging-setup.png
    ...   css=#content

```



adapt-and-extend/config/../../_robot/imaging-setup.png

Here you can define what scales of images will be available to content editors. Plone automatically creates these image scales on the fly.

The value '88' for image quality is a good compromise between speed and quality.

Markup

```

*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')

```

```
...          Setup Plone site  ${FIXTURE}
Run keyword if sys.argv[0].startswith('bin/robot')
...          Open test browser
Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
  Import library  Remote  ${PLONE_URL}/RobotRemote

  Run keyword if  sys.argv[0].startswith('bin/robot')
  ...            Remote ZODB SetUp  ${FIXTURE}

  ${language} = Get environment variable  LANGUAGE  'en'
  Set default language  ${language}

  Enable autologin as  Manager
  ${user_id} = Translate  user_id
  ...  default=jane-doe
  ${user_fullname} = Translate  user_fullname
  ...  default=Jane Doe
  Create user  ${user_id}  Member  fullname=${user_fullname}
  Set autologin username  ${user_id}

Test Teardown
  Run keyword if  sys.argv[0].startswith('bin/robot')
  ...            Remote ZODB TearDown  ${FIXTURE}

Suite Teardown
  Run keyword if  not sys.argv[0].startswith('bin/robot')
  ...            Teardown Plone Site
  Run keyword if  sys.argv[0].startswith('bin/robot')
  ...            Close all browsers
```

```
*** Test Cases ***

Show Markup setup screen
  Go to  ${PLONE_URL}/@@markup-controlpanel
  Capture and crop page screenshot
  ...  ${CURDIR}/../../_robot/markup-setup.png
  ...  css=#content
```



adapt-and-extend/config/../../_robot/markup-setup.png

“Markup” languages like [Markdown](#) and [RestructuredText](#) are popular with some users, since they can be written using a keyboard without a mouse or pointing device.

Plone allows these markup languages to be used as text input alternatives, if you so desire.

Content Settings

```

*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Open test browser
    Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
    Import library  Remote  ${PLONE_URL}/RobotRemote

    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Remote ZODB SetUp  ${FIXTURE}

    ${language} = Get environment variable  LANGUAGE  'en'
    Set default language  ${language}

    Enable autologin as  Manager
    ${user_id} = Translate  user_id
    ...  default=jane-doe
    ${user_fullname} = Translate  user_fullname
    ...  default=Jane Doe
    Create user  ${user_id}  Member  fullname=${user_fullname}
    Set autologin username  ${user_id}

Test Teardown
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Remote ZODB TearDown  ${FIXTURE}

Suite Teardown
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Teardown Plone Site
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Close all browsers

```

```

*** Test Cases ***

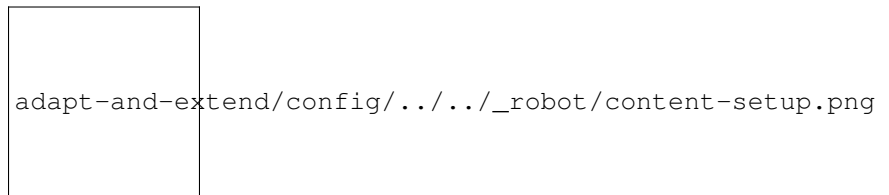
```

```
Show Content setup screen
Go to  ${PLONE_URL}/@@content-controlpanel
Capture and crop page screenshot
...   ${CURDIR}/../../../../_robot/content-setup.png
...   css=#content

Click element  type_id

Select From List  name=type_id  Document

Capture and crop page screenshot
...   ${CURDIR}/../../../../_robot/content-document.png
...   css=#content
```



This seemingly innocent looking screen is the gateway to setting up advanced functionality: Workflows, visibility and versioning settings.

When (Default) is selected, as in the above screenshot, you set the workflow that new content types will get. It is, so to say, the default one for your entire website.

Plone comes with a few pre-defined workflows:

Simple publication workflow This is the default: Items start out as “private”, and then can get published.

Single State workflow The most simple one: everything is always published. Recommended for really simple sites, basically it means there is no real workflow.

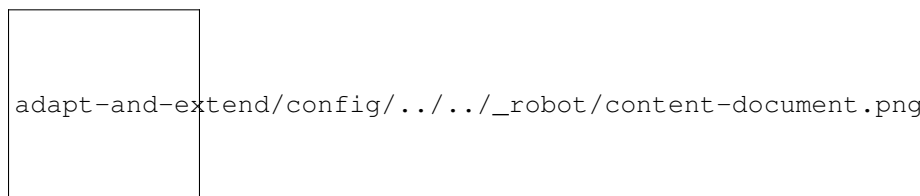
Community workflow Best for a community-driven website, where all members of the community can create content that will be visible, but certain content can be ‘promoted’, usually to the front page.

Intranet/Extranet workflow Meant for intranets, where the majority of users is logged in. Some content can be ‘externally published’ so it is available to anonymous visitors as well.

No Workflow In this one, items ‘inherit’ the state of their parent folder. If a certain type of content has the “No Workflow” workflow, it will be published if the folder where it lives is published, and private if the folder where it lives is private.

All of these workflows can be assigned on a per-contenttype base.

Once you select a content type in the top drop-down field, more options become available. In the screenshot below, we have picked the “Page” content type:



Now we can set a whole range of options on the “Page” content type:

Globally addable Selecting this option means the content type can be added anywhere in the site (*provided you or somebody else has not set up specific exclusions for an individual Folder*)

Allow comment This setting **overrides** the global setting in the [discussion settings](#)

Visible in searches Will this content type show up in search results?

Versioning policy The “Page” content type has ‘Automatic’ versioning enabled, meaning you can use the [versioning features](#) on it.

Manage portlets assigned to this content type This link will take you to “content-specific” portlets. This is a feature that some add-ons use.

Current workflow: Explains in short what the current workflow for this content type does

New workflow: Allows you to change the workflow for this content type.

All in all, this allows sophisticated setups, where some content items just follow the Plone standard workflow, but some others (think of a content type for Expenditures) goes through a whole other chain of workflow states.

Note: Defining your own workflows

It is no problem to define your own workflows. By default, however, that involves a trip to the [Management Interface](#) which is not incredibly user friendly, and defining new workflows is a task best left for your Site Administrator or other specialist.

There are several add-ons, however, like [plone.app.workflowmanager](#) that will present a graphical tool to define workflows.

Dexterity Content Types

```
*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Open test browser
    Run keyword and ignore error  Set window size  @{DIMENSIONS}
```

```
Test Setup
    Import library    Remote    ${PLONE_URL}/RobotRemote

    Run keyword if    sys.argv[0].startswith('bin/robot')
    ...              Remote ZODB SetUp    ${FIXTURE}

    ${language} = Get environment variable    LANGUAGE    'en'
    Set default language    ${language}

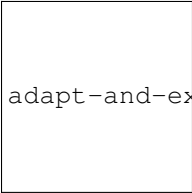
    Enable autologin as    Manager
    ${user_id} = Translate    user_id
    ...    default=jane-doe
    ${user_fullname} = Translate    user_fullname
    ...    default=Jane Doe
    Create user    ${user_id}    Member    fullname=${user_fullname}
    Set autologin username    ${user_id}

Test Teardown
    Run keyword if    sys.argv[0].startswith('bin/robot')
    ...              Remote ZODB TearDown    ${FIXTURE}

Suite Teardown
    Run keyword if    not sys.argv[0].startswith('bin/robot')
    ...              Teardown Plone Site
    Run keyword if    sys.argv[0].startswith('bin/robot')
    ...              Close all browsers
```

```
*** Test Cases ***
```

```
Show Dexterity setup screen
    Go to    ${PLONE_URL}/@@dexterity-types
    Capture and crop page screenshot
    ...    ${CURDIR}/../../../../_robot/dexterity-setup.png
    ...    css=#content
```



adapt-and-extend/config/../../../../_robot/dexterity-setup.png

TODO:

Documentation for TTW (*“through the web”*) configuration of dexterity content types is under development.

Managing Users and Groups

```
*** Settings ***
```

```
Resource    plone/app/robotframework/server.robot
Resource    plone/app/robotframework/keywords.robot
Resource    Selenium2Screenshots/keywords.robot
```

```
Library    OperatingSystem
```

```

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...            Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...            Open test browser
    Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
    Import library  Remote  ${PLONE_URL}/RobotRemote

    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...            Remote ZODB SetUp  ${FIXTURE}

    ${language} =  Get environment variable  LANGUAGE  'en'
    Set default language  ${language}

    Enable autologin as  Manager
    ${user_id} =  Translate  user_id
    ...  default=jane-doe
    ${user_fullname} =  Translate  user_fullname
    ...  default=Jane Doe
    Create user  ${user_id}  Member  fullname=${user_fullname}
    Set autologin username  ${user_id}

Test Teardown
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...            Remote ZODB TearDown  ${FIXTURE}

Suite Teardown
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...            Teardown Plone Site
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...            Close all browsers

```

```

*** Test Cases ***

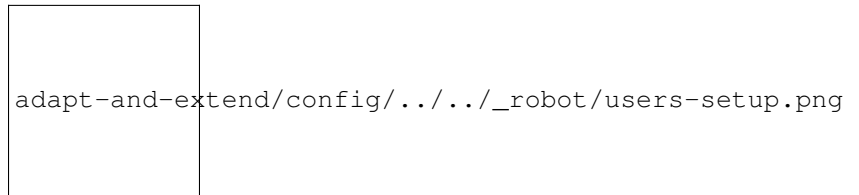
Show Users setup screen
    Go to  ${PLONE_URL}/@@usergroup-userprefs
    Capture and crop page screenshot
    ...  ${CURDIR}/../../_robot/users-setup.png
    ...  css=#content

    Go to  ${PLONE_URL}/@@usergroup-groupprefs
    Capture and crop page screenshot
    ...  ${CURDIR}/../../_robot/groups-setup.png

```

```
... css=#content
Go to  ${PLONE_URL}/@@usergroup-controlpanel
Capture and crop page screenshot
...  ${CURDIR}/../../../../_robot/users-settings.png
... css=#content
Go to  ${PLONE_URL}/@@member-fields
Capture and crop page screenshot
...  ${CURDIR}/../../../../_robot/users-fields.png
... css=#content
```

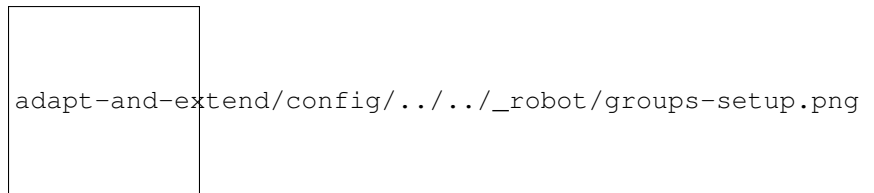
User setup



Here you can search for existing users, and add a new user. When you search for existing users you can edit their properties (if you have the Manager role)

Note: Many large organisations use a form of centralised user management, such as Active Directory or LDAP. In that case, it may not be possible to define users and/or groups here.

Group setup



This gives an overview of the current groups, and allows you to define new ones. Also sets the security mapping: here you map Groups to Roles.

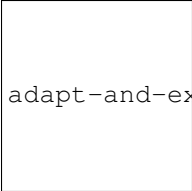
As also seen in the guide on [Sharing and collaborating](#) it's useful to connect Roles (who have permissions) to Groups rather than to individual users. This will make site maintenance easier.

Settings for many users/groups

These options will simply optimize the presentation of users and groups according to the size of your website. If you have thousands or hundreds of thousands of users, you don't want to see them in a list. You will be presented with a searchbox instead.

Member data (information about users)

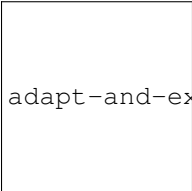
Here you can define what information you want to store about users. There is a range of fields to choose from, from images to numbers to multiple choice. Each of these fields can be made *required* or not.



adapt-and-extend/config/../../../../_robot/users-settings.png

Examples could be

- a “choice” field for dietary preferences and restrictions
- Twitter handle
- whatever else you can think of, or your organisation requires!



adapt-and-extend/config/../../../../_robot/users-fields.png

Enabling HTML embed codes

Description

You can set up Plone so it will not allow you to paste the code necessary to embed videos, slideshows or music players from popular websites such as Flickr, YouTube, Google Maps and MySpace. Learn how to adjust the HTML filtering to achieve the desired level of safety versus convenience.

```
*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...             Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')
```

```
...          Open test browser
Run keyword and ignore error Set window size  @{DIMENSIONS}

Test Setup
  Import library  Remote  ${PLONE_URL}/RobotRemote

  Run keyword if  sys.argv[0].startswith('bin/robot')
  ...          Remote ZODB SetUp  ${FIXTURE}

  ${language} = Get environment variable  LANGUAGE  'en'
  Set default language  ${language}

  Enable autologin as  Manager
  ${user_id} = Translate  user_id
  ...  default=jane-doe
  ${user_fullname} = Translate  user_fullname
  ...  default=Jane Doe
  Create user  ${user_id}  Member  fullname=${user_fullname}
  Set autologin username  ${user_id}

Test Teardown
  Run keyword if  sys.argv[0].startswith('bin/robot')
  ...          Remote ZODB TearDown  ${FIXTURE}

Suite Teardown
  Run keyword if  not sys.argv[0].startswith('bin/robot')
  ...          Teardown Plone Site
  Run keyword if  sys.argv[0].startswith('bin/robot')
  ...          Close all browsers
```

```
*** Test Cases ***

Show HTML filter setup screen
  Go to  ${PLONE_URL}/@@filter-controlpanel
  Capture and crop page screenshot
  ...  ${CURDIR}/../../../../_robot/filter-setup.png
  ...  css=#content
```



adapt-and-extend/config/../../../../_robot/filter-setup.png

Important security note

Making these configuration changes has serious security implications for your site.

Plone filters out many tags for a good reason: they can be abused by your site users to create privilege escalation attacks.

If you have allowed untrusted people to create content on your Plone site, then a malicious person could create some “nasty” JavaScript in some content, then trick a person with Admin rights into viewing that content. That “nasty” JavaScript can now do HTTP requests to interact with the Plone site with the full Admin rights granted to the trusted user.

Bottom line: do not use this technique to enable embeddable content in your Plone site unless you are certain that you trust all users who are allowed to create content in your site.

Plone 5

In Plone 5, there are two steps you need to take in order to embed content that is not using an *iframe* tag:

Note: Per default, Plone 5 will allow `<iframe>` as a valid tag. That enables embedding media from the most popular sites like Vimeo and YouTube.

This behavior is a change from earlier versions. If you are in a high-security environment, simply add “iframe” to the list of *nasty tags* and embedding will stop working.

First, go to Site Setup>TinyMCE Visual Editor then click on the Toolbar tab.

- Enable the checkbox next to “Insert/edit Media”
- Scroll down to the bottom of the screen and click “Save”

Then, go to Site Setup>HTML Filtering

- Remove “Object” and “Embed” from the “Nasty Tags” list
- Remove “Object” and “Param” from the “Stripped Tags” list
- Add “Embed” to the “Custom Tags” list
- Scroll down to the bottom of the screen and click “Save”

With these changes made, you should be able to click newly-added “Embed Media” button in the TinyMCE toolbar. You can paste in the URL of a YouTube video, and TinyMCE will do the rest for you!

For a Flickr slideshow, and most other embeds, switch into HTML editing mode and paste in the raw embed code.

Note: To allow completely arbitrary HTML codes, see [WYSIWYG text editing and TinyMCE](#) and [David Glick’s blogpost](#).

Security settings

```
*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content
```

```
*** Keywords ***

Suite Setup
    Run keyword if    not sys.argv[0].startswith('bin/robot')
    ...               Setup Plone site    ${FIXTURE}
    Run keyword if    sys.argv[0].startswith('bin/robot')
    ...               Open test browser
    Run keyword and ignore error    Set window size    @DIMENSIONS

Test Setup
    Import library    Remote    ${PLONE_URL}/RobotRemote

    Run keyword if    sys.argv[0].startswith('bin/robot')
    ...               Remote ZODB Setup    ${FIXTURE}

    ${language} =    Get environment variable    LANGUAGE    'en'
    Set default language    ${language}

    Enable autologin as    Manager
    ${user_id} =    Translate    user_id
    ...    default=jane-doe
    ${user_fullname} =    Translate    user_fullname
    ...    default=Jane Doe
    Create user    ${user_id}    Member    fullname=${user_fullname}
    Set autologin username    ${user_id}

Test Teardown
    Run keyword if    sys.argv[0].startswith('bin/robot')
    ...               Remote ZODB TearDown    ${FIXTURE}

Suite Teardown
    Run keyword if    not sys.argv[0].startswith('bin/robot')
    ...               Teardown Plone Site
    Run keyword if    sys.argv[0].startswith('bin/robot')
    ...               Close all browsers
```

```
*** Test Cases ***

Show Security setup screen
    Go to    ${PLONE_URL}/@@security-controlpanel
    Capture and crop page screenshot
    ...    ${CURDIR}/../../../../_robot/security-setup.png
    ...    css=#content
```



adapt-and-extend/config/../../../../_robot/security-setup.png

Of the various settings here, the most important ones are:

Enable self-registration Setting this option will mean that new users can register themselves. It is **strongly** advised to keep the “let users select their own passwords” UNchecked when choosing this option, as at least all new users will have to have a valid email. This helps in slowing down bot-generated attacks.

Use email address as login name As most people have difficulty remembering login names, it can be good to allow email addresses as logins.

Error log

```
*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Open test browser
    Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
    Import library  Remote  ${PLONE_URL}/RobotRemote

    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Remote ZODB SetUp  ${FIXTURE}

    ${language} =  Get environment variable  LANGUAGE  'en'
    Set default language  ${language}

    Enable autologin as  Manager
    ${user_id} =  Translate  user_id
    ...  default=jane-doe
    ${user_fullname} =  Translate  user_fullname
    ...  default=Jane Doe
    Create user  ${user_id}  Member  fullname=${user_fullname}
    Set autologin username  ${user_id}

Test Teardown
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Remote ZODB TearDown  ${FIXTURE}

Suite Teardown
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Teardown Plone Site
    Run keyword if  sys.argv[0].startswith('bin/robot')
```

```
... Close all browsers
```

```
*** Test Cases ***

Show Error log setup screen
    Go to  ${PLONE_URL}/prefs_error_log_form
    Capture and crop page screenshot
    ...  ${CURDIR}/../../../../_robot/errorlog-setup.png
    ...  css=#content
```



adapt-and-extend/config/../../../../_robot/errorlog-setup.png

From this screen you can see the most recent errors and *exceptions*.

Remember: not all ‘exceptions’ are harmful or indications of potential problems. It can be that visitors just mis-typed a URL, or many other reasons.

But in case of trouble, or if you are trying to find out if your permissions are working correctly, this should be your first port of call.

Maintenance and packing

```
*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...            Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...            Open test browser
    Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
```

```

Import library    Remote    ${PLONE_URL}/RobotRemote

Run keyword if    sys.argv[0].startswith('bin/robot')
...              Remote ZODB SetUp    ${FIXTURE}

${language} =    Get environment variable    LANGUAGE    'en'
Set default language    ${language}

Enable autologin as    Manager
${user_id} =    Translate    user_id
...    default=jane-doe
${user_fullname} =    Translate    user_fullname
...    default=Jane Doe
Create user    ${user_id}    Member    fullname=${user_fullname}
Set autologin username    ${user_id}

Test Teardown
Run keyword if    sys.argv[0].startswith('bin/robot')
...              Remote ZODB TearDown    ${FIXTURE}

Suite Teardown
Run keyword if    not sys.argv[0].startswith('bin/robot')
...              Teardown Plone Site
Run keyword if    sys.argv[0].startswith('bin/robot')
...              Close all browsers

```

```

*** Test Cases ***

Show ZODB maintenance setup screen
Go to    ${PLONE_URL}/@@maintenance-controlpanel
Capture and crop page screenshot
...    ${CURDIR}/../../../../_robot/zodb-setup.png
...    css=#content

```



adapt-and-extend/config/../../../../_robot/zodb-setup.png

As you can see, there is not much to do here with Plone manager permissions.

If you have *Zope manager* permissions, there is one extra option here: to *pack* your database. However, in any normal deploying setup, you would want to automate that task via a cronjob or similar mechanism.

The Management Interface

Here be dragons.

The Management Interface is a direct interface into the backend software stack of Plone. While it can still serve as a valuable tool for Plone specialists to fix problems or accomplish certain tasks, it is *not* recommended as a regular tool for Plone maintenance.

If you are just starting out, this is probably not where you want to go.

If you are a Plone expert, this is where to get to the venerable Management Interface.

Note: Only users with role “Manager” can use this option.

Caching

Change settings

Import settings

RAM cache

i

Info

First time here? We recommend that you get started by importing a preconfigured set of caching rules.

Caching settings

Control how pages, images, style sheets and other resources are cached.

Global settings

Caching proxies

In-memory cache

Caching operations

Detailed settings

☐ **Enable caching**
If this option is disabled, no caching will take place.

Save

Cancel

Caching is the process where information is kept in a temporary store, to deliver it to the visitor more quickly.

It is always a balancing act between the ‘freshness’ of the content, and speed of display.

Enabling caching here within Plone is highly recommended, but fine-tuning it can be more of an art than a science.

Plone comes with a fairly conservative, but highly effective set of defaults. Importing those settings is your best course of action in almost all cases.

Plone’s internal caching works even better when used together with an external cache, such as Varnish.

See the [Guide to caching](#) for more information.

Configuration Registry

Users with role “Manager” can directly see, and change, many variables that influence Plone.

As you can see, there is a large number of these, and finding the right one is quicker done by filtering or selecting a prefix. (Add-on packages will also create their own prefix)

Note: Setting variables directly here is *not* recommended as part of regular maintenance. It is a useful tool for *inspecting* variables for expert users.

```
*** Settings ***  
  
Resource  plone/app/robotframework/server.robot
```



```

Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Open test browser
    Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
    Import library  Remote  ${PLONE_URL}/RobotRemote

    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Remote ZODB SetUp  ${FIXTURE}

    ${language} =  Get environment variable  LANGUAGE  'en'
    Set default language  ${language}

    Enable autologin as  Manager
    ${user_id} =  Translate  user_id
    ...  default=jane-doe
    ${user_fullname} =  Translate  user_fullname
    ...  default=Jane Doe
    Create user  ${user_id}  Member  fullname=${user_fullname}
    Set autologin username  ${user_id}

Test Teardown
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Remote ZODB TearDown  ${FIXTURE}

Suite Teardown
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...              Teardown Plone Site
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...              Close all browsers

```

```

*** Test Cases ***

Show Configuration Registry screen
    Go to  ${PLONE_URL}/portal_registry
    Capture and crop page screenshot
    ...  ${CURDIR}/../_robot/configuration-registry.png
    ...  css=#content

```

adapt-and-extend/config/../../_robot/configuration-regist

Resource Registries

```
*** Settings ***

Resource  plone/app/robotframework/server.robot
Resource  plone/app/robotframework/keywords.robot
Resource  Selenium2Screenshots/keywords.robot

Library  OperatingSystem

Suite Setup  Run keywords  Suite Setup  Test Setup
Suite Teardown  Run keywords  Test teardown  Suite Teardown

*** Variables ***

${FIXTURE}  plone.app.robotframework.PLONE_ROBOT_TESTING
@{DIMENSIONS}  1024  768
@{APPLY_PROFILES}  plone.app.contenttypes:plone-content

*** Keywords ***

Suite Setup
    Run keyword if  not sys.argv[0].startswith('bin/robot')
    ...            Setup Plone site  ${FIXTURE}
    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...            Open test browser
    Run keyword and ignore error  Set window size  @{DIMENSIONS}

Test Setup
    Import library  Remote  ${PLONE_URL}/RobotRemote

    Run keyword if  sys.argv[0].startswith('bin/robot')
    ...            Remote ZODB Setup  ${FIXTURE}

    ${language} =  Get environment variable  LANGUAGE  'en'
    Set default language  ${language}

    Enable autologin as  Manager
    ${user_id} =  Translate  user_id
    ...  default=jane-doe
    ${user_fullname} =  Translate  user_fullname
    ...  default=Jane Doe
    Create user  ${user_id}  Member  fullname=${user_fullname}
    Set autologin username  ${user_id}

Test Teardown
```

```

Run keyword if sys.argv[0].startswith('bin/robot')
...           Remote ZODB TearDown  ${FIXTURE}

Suite Teardown
Run keyword if not sys.argv[0].startswith('bin/robot')
...           Teardown Plone Site
Run keyword if sys.argv[0].startswith('bin/robot')
...           Close all browsers

```

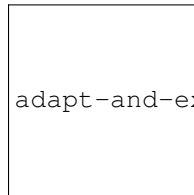
```

*** Test Cases ***

Show Resource Registry screen
Go to  ${PLONE_URL}/@@resourceregistry-controlpanel
Capture and crop page screenshot
...   ${CURDIR}/../../../../_robot/resource-registry.png
...   css=#content
Click link  Less Variables
Capture and crop page screenshot
...   ${CURDIR}/../../../../_robot/less-variables.png
...   css=#content

```

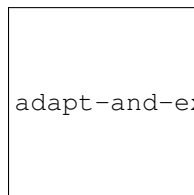
The Resource Registry allows access to JavaScript, CSS and LESS resources.



adapt-and-extend/config/../../../../_robot/resource-registry.png

LESS variables

Particularly useful are the many LESS variables that affect the theme of a site:



adapt-and-extend/config/../../../../_robot/less-variables.png

Installing Add-ons

See *this section*

Custom Content Types

The recommended way to develop new custom content types is by using *Dexterity*. You'll find the full manual [here](#).

Up until Plone 4.1, the standard way was to use *Archetypes*. You can still develop using Archetypes, and they will remain functional. For any **new** development, Dexterity is strongly recommended.

The Archetypes reference manual will remain available at the [Plone 4 documentention](#)

Installing, Managing and Updating Plone

Installing Plone

Plone Installation Requirements

Description

Requirements for installing Plone. Details the tools and libraries (dependencies) required to install Plone.

Hosting requirements

To run a Plone based web site on your own server you need:

- A server machine connected to Internet (public sites) or your intranet (company intranet sites);
- Remote console, like SSH access, for installing Plone. FTP is not enough.

Plone requires several system libraries. These need to be installed by a user with root access. If you would like to install Plone using a consumer hosting service, you must ensure that the service includes SSH and root access.

Operating system

Plone has been successfully installed on:

- Nearly every popular Linux distribution;
- Nearly every popular BSD distribution
- OS X (using our OS X installer or XCode)
- Solaris and several other proprietary *nix systems
- Windows

Hardware (or virtual environment) requirements

The hardware requirements below give a rough estimate of the minimum hardware setup needed for a Plone server.

Add-on products and caching solutions may increase RAM requirements.

A single Plone installation is able to run many Plone sites. This makes it easy to host multiple sites on the same server.

Plone runs on:

- Raspberry Pi
- Chromebooks
- Windows PCs
- Macs
- servers
- containers such as Docker
- virtual machines such as Vagrant
- cloud services such as Amazon, Rackspace, and Linode

Minimum requirements

- Minimum 256 MB RAM and 512 MB of swap space per Plone site
- Minimum 512 MB hard disk space

Recommended

- 2 GB or more RAM per Plone site
- 40 GB or more hard disk space

All Plone versions

What follows is an overview of Plone’s overall software requirements. Each Plone installer (Unified Installer, Vagrant/VirtualBox, Windows buildout) will manage its dependencies and requirements differently.

Windows

Plone requires Python and Visual C++.

UNIX-based platforms

Plone requires Python and a complete GNU build kit including GCC including gcc, gmake, patch, tar, gunzip, bunzip2, wget.

Most required libraries listed below must be installed as development versions (dev).

Tools and libraries marked with “*” are either included with the Unified Installer or automatically downloaded.

If you use your system Python, you should use Python's `virtualenv` to create an isolated virtual Python. System Pythons may use site libraries that will otherwise interfere with Zope/Plone.

Optional libraries

If Plone can find utilities that convert various document formats to text, it will include them in the site index. To get PDFs and common office automation formats indexed, add:

- `poppler-utils` (PDFs)
- `wv` (office docs)

These may be added after initial installation.

Plone 5

Python

Python 2.7 (dev), built with support for `expat` (`xml.parsers.expat`), `zlib` and `ssl`. (Python XML support may be a separate package on some platforms.)*

`virtualenv`*

Libraries

- `libz` (dev)
- `libjpeg` (dev)*
- `readline` (dev)*
- `libexpat` (dev)
- `libssl` or `openssl` (dev)
- `libxml2` $\geq 2.7.8$ (dev)*
- `libxslt` $\geq 1.1.26$ (dev)*

You may also need to install dependencies needed by `Pillow` a fork of the Python Image Library. For further information please read: <https://pillow.readthedocs.org/en/latest/installation.html>

Installation

Introduction

This document covers the basics of installing Plone on popular operating systems. It will also point you to other documents for more complex or demanding installations.

Plone runs as an application on the Zope application server. That server is installed automatically by the install process.

Warning: We strongly advise against installing Plone via OS packages or ports.

There is no `.rpm`, `.deb`, or BSD port that is supported by the Plone community.

Plone dependencies can and should be installed via package or port – but not Plone itself.

Note: For installing and running Plone on CentOS, please make sure that you have enough Memory and Swap. If you use the Unified Installer on CentOS, make sure that you have at least 1024 MB of Memory.

Building lxml is resource consuming.

Download Plone

Plone is available for Mac OSX X, Linux and BSD operating systems. For Windows, we currently advise running Plone 5 in a virtualmachine or Vagrant image.

We anticipate having a binary windows installer for later releases.

[Download the latest Plone release.](#)

From here, you can also find links to the Vagrant install kit (if you wish to install Plone for evaluation or development on a Windows, OS X or any other machine that supports VirtualBox and Vagrant).

Installation on Linux, BSD and other Unix workalikes requires a source code installation, made easy by our Unified Installer. “Unified” refers to its ability to install on most Unix workalikes.

Plone installation requirements

See [Plone installation requirements](#) for detailed requirements.

- You need at least a dedicated or virtual private server (VPS) with 512 MB RAM available. Shared hosting is not supported unless the shared hosting company says Plone is good to go.

How to install Plone

Plone can run on all popular desktop or server operating systems, including Linux, OS X, BSD and Microsoft Windows. (Note: currently there is no binary installer for Plone 5 on Windows, we recommend using the [Vagrant kit](#))

- You can install Plone on a server for production usage
- You can install Plone locally on your own computer for development and test drive

Installing Plone using the Unified UNIX Installer

Note: Running Plone in **production** will normally also entail other software for optimal security and performance, like a front-end webserver, caching, and firewall.

See [Deploying and installing Plone in production](#) , and you may also be interested in automated full-stack deployment.

This recipe is good for:

- Plone development and testing on Ubuntu / Debian
- Operating system installations where you have administrator (root) access. Note that root access is not strictly necessary as long as you have required software installed beforehand on the server, but this tutorial assumes you need to install the software yourself and you are the admin. If you don't have the ability to install system libraries, you'll need to get your sysadmin to do it for you. The libraries required are in common use.

The resulting installation is self-contained, does not touch system files, and is safe to play with (no root/sudoing is needed).

If you are not familiar with UNIX operating system command line you might want to study this [Linux shell tutorial](#) first.

Install the operating system software and libraries needed to run Plone

Note: If the `sudo` command is not recognized or does not work you don't have administrator rights on your operating system.

Please contact your server vendor or consult the operating system support forum.

You will probably also want these optional system packages for handling of PDF and Office files:

Note: `libreadline-dev` or `readline-devel` is only necessary if you wish to build your own python rather than use your system's python 2.7.

If you're planning on developing with Plone, install git version control support

Download the latest Plone unified installer

Download from the [plone.org download page](#) to your server using `wget` command.

Curl also works.

Substitute the latest version number for 5.0 in the instructions below.

```
wget --no-check-certificate https://launchpad.net/plone/5.0/5.0.4/+download/Plone-5.0.4-UnifiedInstaller.tgz
```

Run the Plone installer in standalone mode

Extract the downloaded file

```
tar -xf Plone-5.0.4-UnifiedInstaller.tgz
```

Go the folder containing installer script

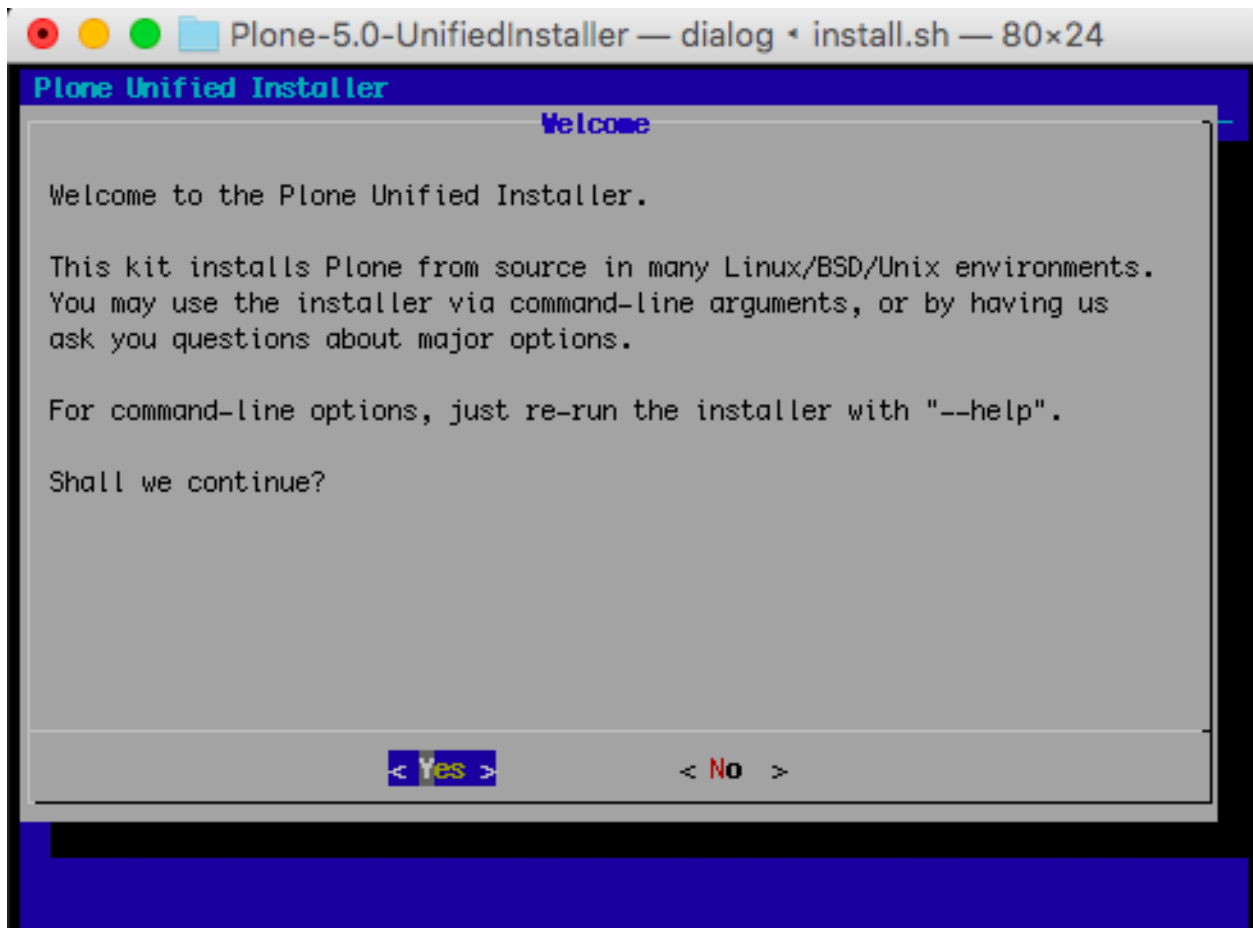
```
cd Plone-5.0.4-UnifiedInstaller
```

Note: This will run the installer without any extra options, like setting passwords, setting the install path or anything else, for a full overview over all the possible options use `./install.sh --help`.

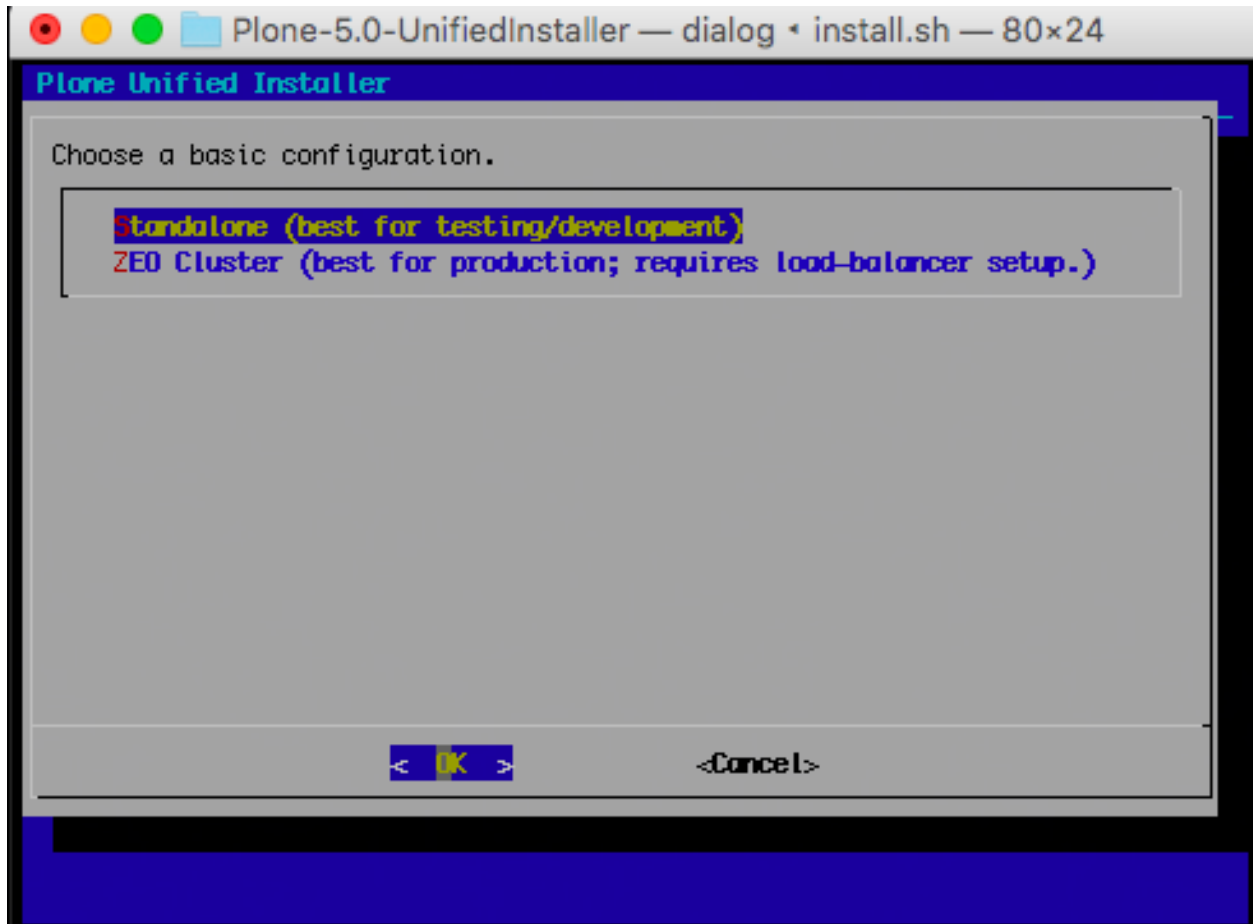
Run script

```
./install.sh
```

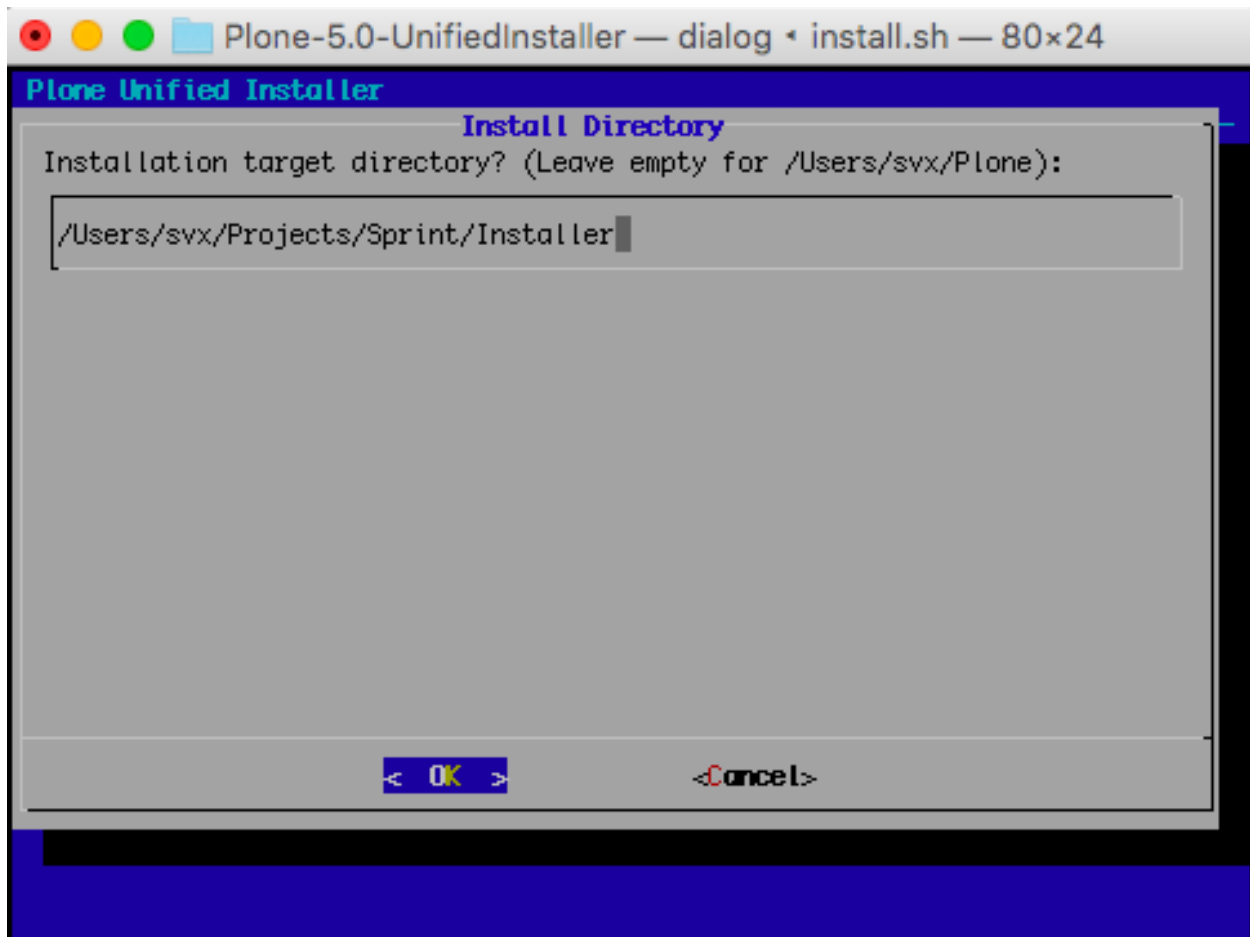
Please follow the instructions on the screen



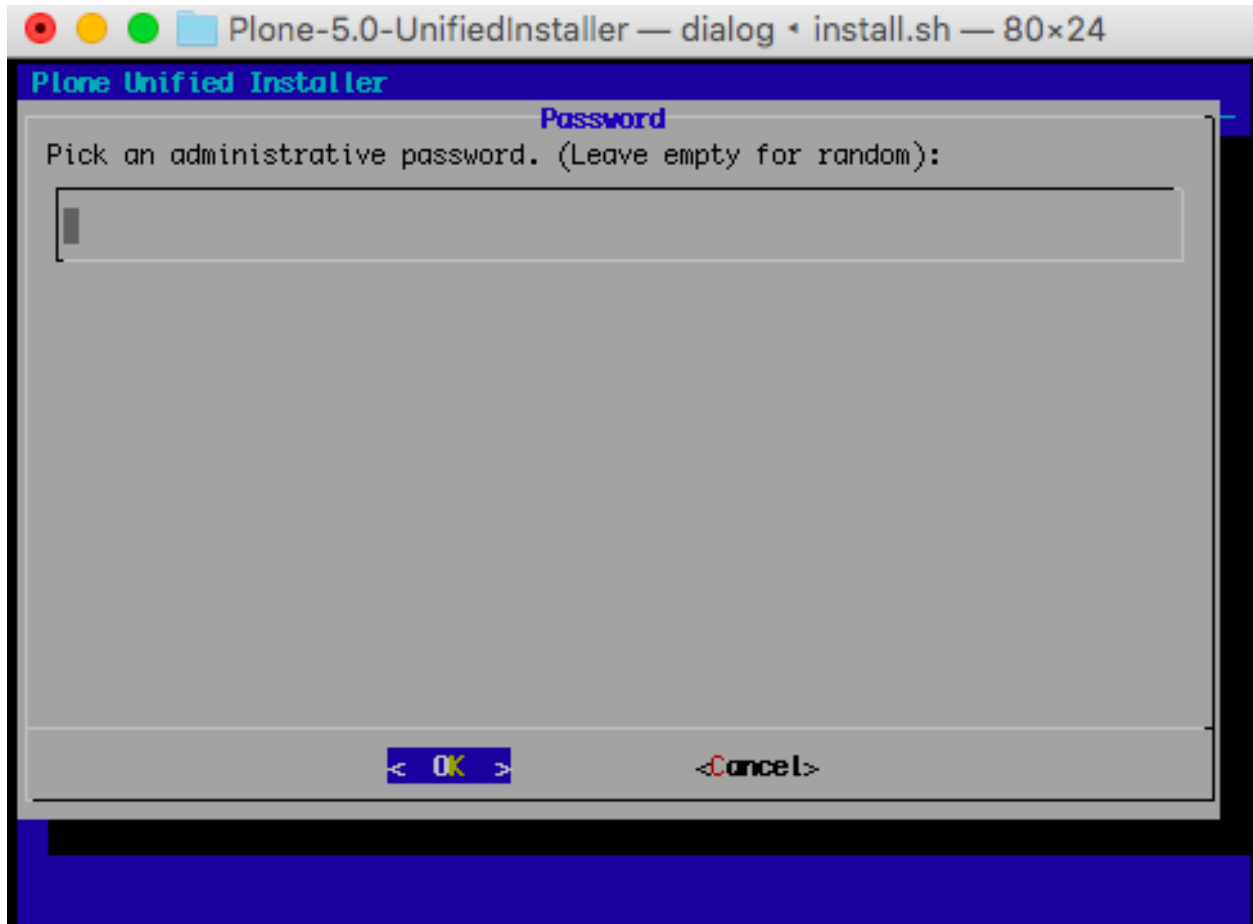
We choose here for the `standalone` mode



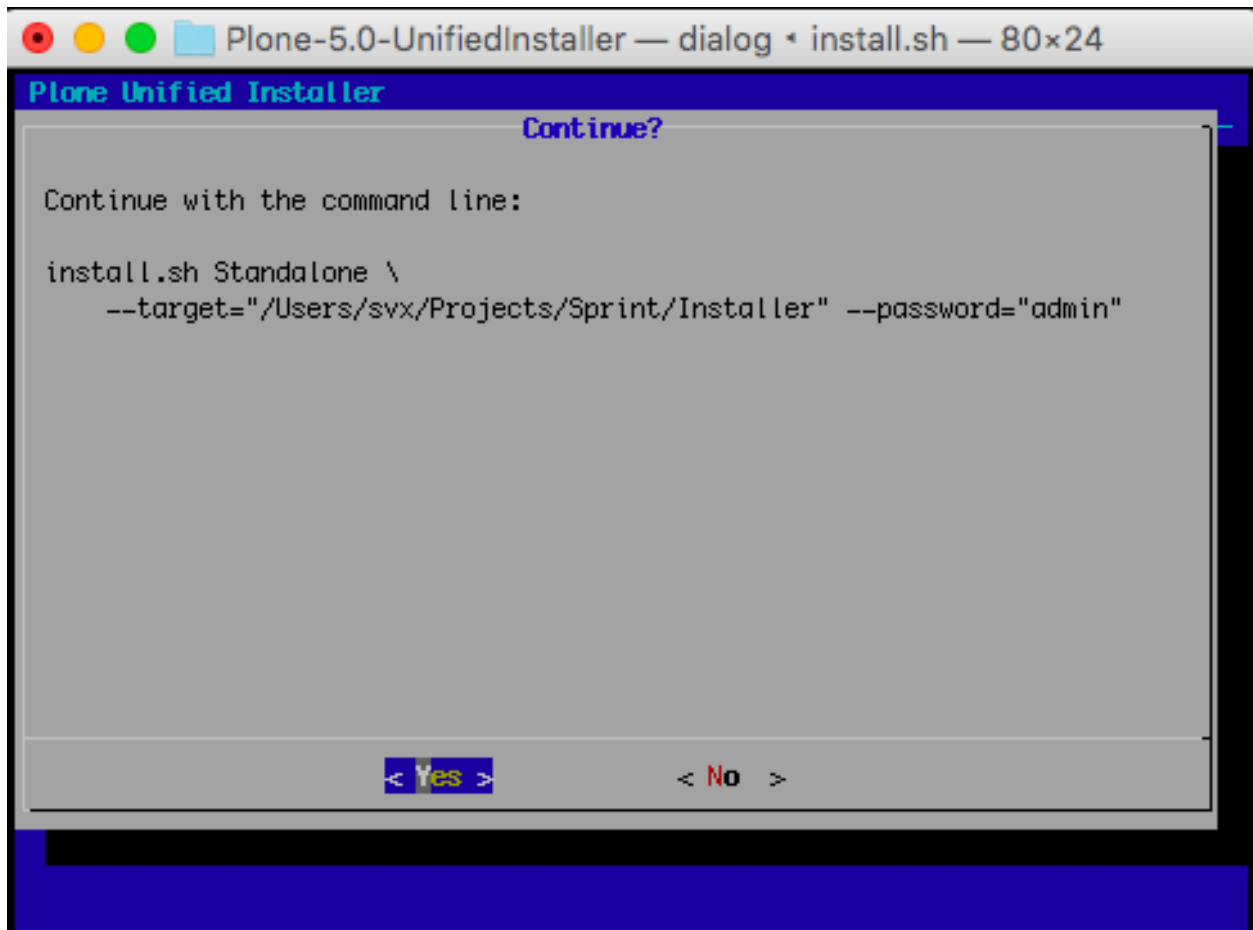
Accept the default installation target or change the path



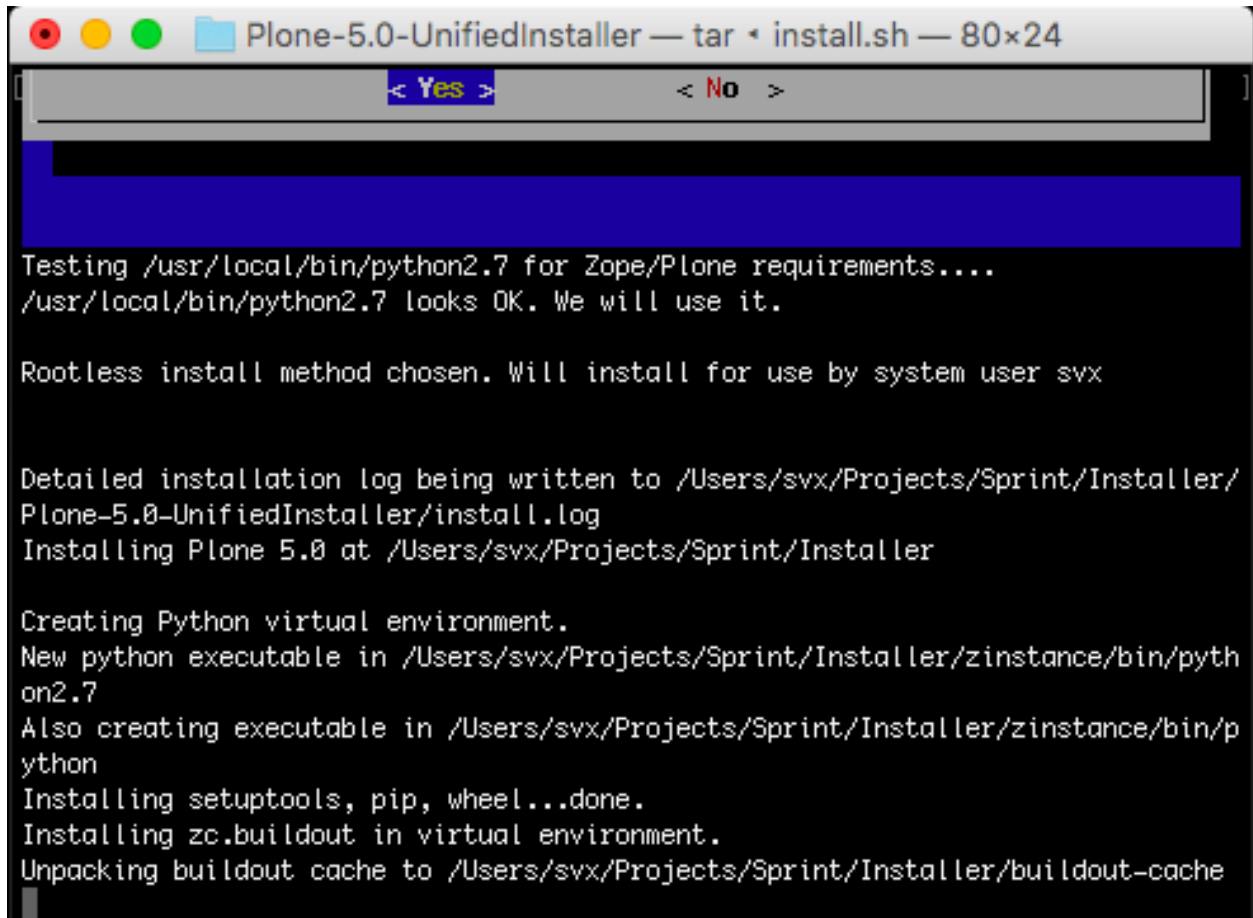
Choose a password option



Control the settings to make sure everything is as you want it



Wait till the installer is done



A terminal window titled "Plone-5.0-UnifiedInstaller — tar • install.sh — 80x24". The window has a standard macOS title bar with red, yellow, and green window control buttons. Below the title bar is a grey bar with navigation controls: "< Yes >" (highlighted in blue) and "< No >". The main area of the terminal is black with white text. It shows the output of the installation script, including a confirmation prompt, testing of Python 2.7, selection of the rootless install method, logging, and the creation of a virtual environment with various tools.

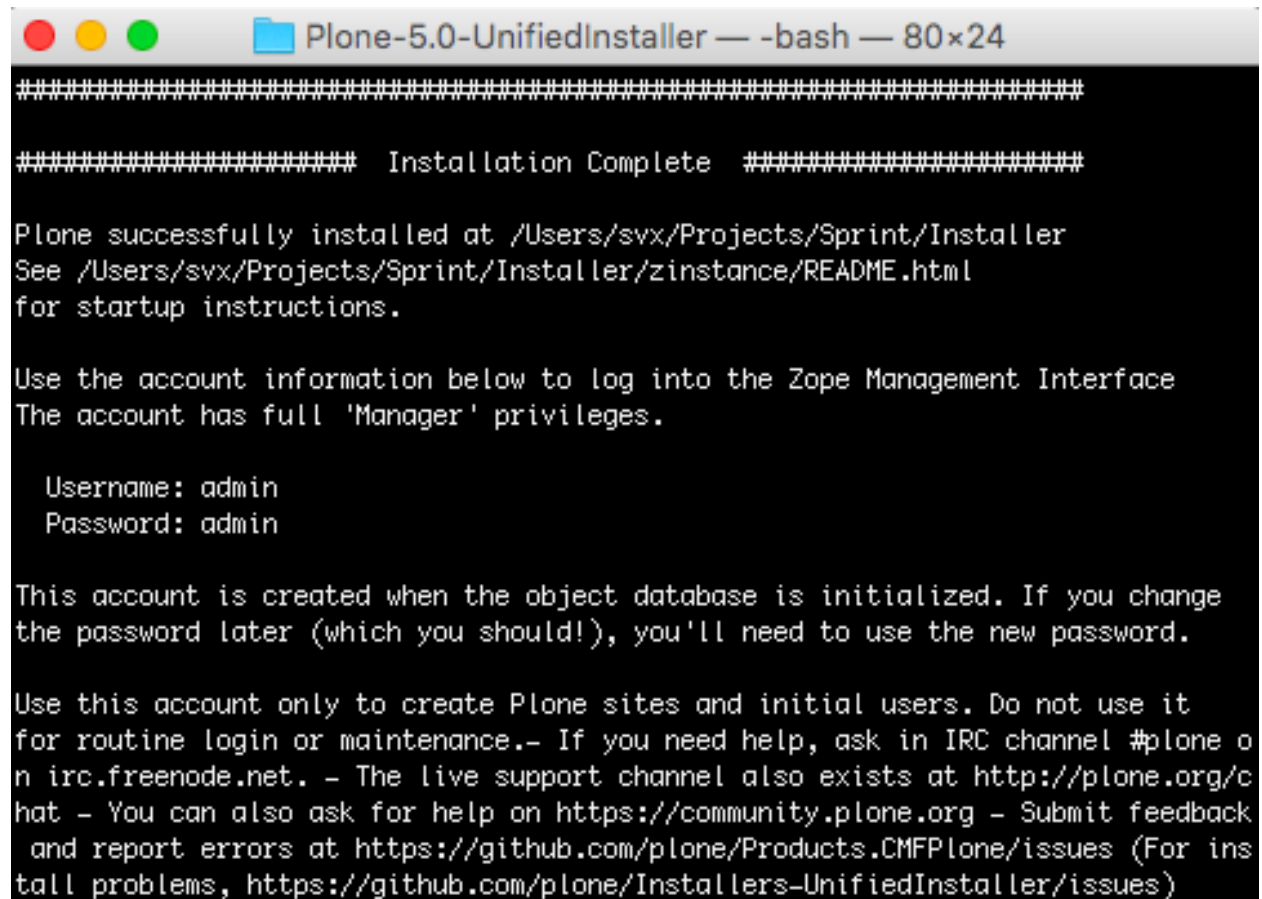
```
< Yes > < No >

Testing /usr/local/bin/python2.7 for Zope/Plone requirements....
/usr/local/bin/python2.7 looks OK. We will use it.

Rootless install method chosen. Will install for use by system user svx

Detailed installation log being written to /Users/svx/Projects/Sprint/Installer/
Plone-5.0-UnifiedInstaller/install.log
Installing Plone 5.0 at /Users/svx/Projects/Sprint/Installer

Creating Python virtual environment.
New python executable in /Users/svx/Projects/Sprint/Installer/zinstance/bin/pyth
on2.7
Also creating executable in /Users/svx/Projects/Sprint/Installer/zinstance/bin/p
ython
Installing setuptools, pip, wheel...done.
Installing zc.buildout in virtual environment.
Unpacking buildout cache to /Users/svx/Projects/Sprint/Installer/buildout-cache
```

A terminal window titled "Plone-5.0-UnifiedInstaller — -bash — 80x24" displays the output of the Plone 5.0 UnifiedInstaller. The output is as follows:

```
#####  
##### Installation Complete #####  
  
Plone successfully installed at /Users/svx/Projects/Sprint/Installer  
See /Users/svx/Projects/Sprint/Installer/zinstance/README.html  
for startup instructions.  
  
Use the account information below to log into the Zope Management Interface  
The account has full 'Manager' privileges.  
  
Username: admin  
Password: admin  
  
This account is created when the object database is initialized. If you change  
the password later (which you should!), you'll need to use the new password.  
  
Use this account only to create Plone sites and initial users. Do not use it  
for routine login or maintenance.- If you need help, ask in IRC channel #plone o  
n irc.freenode.net. - The live support channel also exists at http://plone.org/c  
hat - You can also ask for help on https://community.plone.org - Submit feedback  
and report errors at https://github.com/plone/Products.CMFPlone/issues (For ins  
tall problems, https://github.com/plone/Installers-UnifiedInstaller/issues)
```

The default admin credentials will be printed to the console, and saved in the file `adminPassword.txt` in the resulting install. You can change this password after logging in to the Management Interface.

Note: The password is also written down in the `buildout.cfg` file, but this setting is not effective after Plone has been started for the first time. Changing this setting does not do any good.

Install the Plone developer tools

If you're using this Plone install for development, add the common development tool set.

```
cd ~/Plone/zinstance  
bin/buildout -c develop.cfg
```

You'll need to add the `“-c develop.cfg”` again each time you run `buildout`, or you'll lose the extra development tools.

Start Plone

If you're developing, start Plone in foreground mode for a test run (you'll see potential errors in the console):

```
cd ~/Plone/zinstance  
bin/plonectl fg
```


When you start Plone in the foreground, it runs in debug mode, which is much slower than production mode since it reloads templates for every request.

For evaluation, instead use:

```
cd ~/Plone/zinstance
bin/plonectl start
```

Use

```
cd ~/Plone/zinstance
bin/plonectl stop
```

to stop the instance.

By default, Plone will listen to port 8080 on available network interfaces. The port may be changed by editing `buildout.cfg` and re-running `buildout`.

You've got Plone

Now take a look at your Plone site by visiting the following address in your webbrowser:

```
http://yourserver:8080
```

The greeting page will let you create a new site. For this you need the login credentials printed to your terminal earlier, also available at `~/Plone/zinstance/adminPassword.txt`.

If everything is OK, press `CTRL-C` in the terminal to stop Plone if you're running in debug mode. Use the `plonectl stop` command if you didn't start in debug mode.

If you have problems, please see the [help guidelines](#).

For automatic start-up when your server boots up, init scripts, etc. please see the [deployment guide](#).

Installing Plone using RPMs, .dev, ... packages

Not supported by Plone community. Plone dependencies can and should be installed via your operating system package manager, to profit from security updates and maintenance, but not Plone itself. The packages that have been offered in the past via `apt`, `yum`, `dnf`, `port` etcetera tend to be unmaintained, old and unsuitable.

Microsoft Windows

For Plone 5, there currently is no binary installer. We recommend using the [Vagrant kit](#)

We anticipate offering a binary installer for Windows at a later moment.

For the Plone 4.3 series, there is a [binary installer](#).

If you wish to develop Plone on Windows you need to set-up a working MingW environment (this can be somewhat painful if you aren't used to it).

OSX

This is the recommended method if you want to try Plone for the first time.

Please use the installer from the download page <http://plone.org/products/plone/releases>.

Installation via the Unified Installer or buildout is similar to Unix. However, you will need to install a command-line build environment. To get a free build kit from Apple, do one of the following:

- Download gcc and command-line tools from <https://developer.apple.com/downloads/>. This will require an Apple developer id.
- Install Xcode from the App Store. After installation, visit the Xcode app's preference panel to download the command-line tools.

After either of these steps, you immediately should be able to install Plone using the Unified Installer.

Proceed as with Linux.

LibXML2/LibXSLT Versions

Don't worry about this if you're using an installer.

Entering debug mode after installation

When you have Plone installed and want to start development you need do *enter debug mode*.

Installer source code

- <https://github.com/plone/Installers-UnifiedInstaller>

Installing add-on packages using buildout

These instructions cover add-on installation process for Plone 5, while mostly being valid for Plone 4 and 3.3 as well. Legacy systems are not covered in these instructions.

Note: Not all add-ons have currently (status: 2015-10-31) been upgraded to work with Plone 5. Take care when trying on add-ons. If an add-on has not yet received a Plone 5-compatible release, it may be that there is already a so-called 'branch' of the sourcecode that is being worked upon, or it may be that the add-on has been superseded. See the "further help" section.

Introduction

Plone uses **Buildout** for installing add-on packages. See *installation instructions* for how to create a Plone installation suitable for development.

Prerequisites

What do you need to know in order to install add-ons for Plone?

- How to use command line of your operating system. This is a hard requirement - you cannot achieve your goal unless you know how to interact with the command line. Here are basics tutorials for [Windows](#) and [Linux](#)
- Working with plain text based configuration files and editing them with a text editor like Notepad
- First create a *development / back-up copy* of your site. Never install to the working production server directly, without first testing the add-on on a test instance.

Discovering Plone add-ons and other python packages

The community maintains a list of Popular Plone [Add-ons](#) .

However, not all Plone packages out there are listed here.

A lot more packages can be found in the [PyPI \(the Python Package index\)](#).

Note: Always check if a third-party add-on is up to date and compatible with your version of Plone. Most packages will have different versions; sometimes their version 1.x is meant for use with Plone 4, and version 2.x is meant for Plone 5. See the documentation of the add-on in question.

Background

Plone installations are managed using [Buildout](#). Plone add-ons are distributed as Python modules, also known as eggs.

- Popular Plone [Add-ons](#) contains a overview about popular add-ons for Plone .
- Add-on file downloads are hosted on the [PyPi Python package repository](#) - along with many other Python software modules.
- the buildout.cfg file in your Plone configuration defines which add-ons are available for your sites to install in Site Setup > Add-ons control panel
- the bin/buildout command (or bin/buildout.exe on Windows) in your Plone installation reads buildout.cfg and automatically downloads required packages when run - you do not need to download any Plone add-ons manually
- Plone site setup -> Add ons control panel defines which add-ons are installed for the current Plone site (remember, there can be many Plone sites on a single Zope application server)

Note: Plone add-ons, though Python eggs, must be installed through buildout as only buildout will regenerate the config files reflecting newly downloaded and installed eggs. Other Python installation methods like easy_install and pip do not apply for Plone add-ons.

Installing add-ons using buildout

Add-on packages which are uploaded to [PyPI](#) can be installed by buildout.

Edit your *buildout.cfg* file and add the add-on package to the list of eggs:

```
[buildout]
...
eggs =
    ...
    Products.PloneFormGen
    collective.supercool
```

Note: The above example works for the buildout created by the unified installer. If you however have a custom buildout you might need to add the egg to the *eggs* list in the *[instance]* section rather than adding it in the *[buildout]* section.

For the changes to take effect you need to re-run buildout from your console:

```
bin/buildout
```

Restart your instance for the changes to take effect:

```
bin/instance restart
```

Pinning add-on versions

As mentioned above, always make sure to test add-ons, and see if you have the right version for your specific version of Plone.

It is **standard, and highly recommended practice** to pick specific versions of add-ons. This practice is called “pinning”.

If you don’t *pin* a specific version, a run of `bin/buildout` might download a newer version of an add-on, that in turn might depend on newer other software. This can lead to breakage of your site.

Therefore, always put the specific version number of the add-on into the section of `buildout.cfg` called “versions”, or into the separate file “versions.cfg” if your buildout has one. An example of version-pinning would be to have:

```
[versions]
Products.PloneFormGen = 1.7.17
collective.supercool = 2.3
```

When *upgrading add-ons* also don’t just upgrade to an unspecified ‘newest’ version, but to a specific newer version that you have previously tested.

Installing development version of add-on packages

If you need to use the latest development version of an add-on package you can get the source in your development installation using the buildout extension `mr.developer`.

‘mr.developer’ can install, or *checkout* from various places: github, gitlab, subversion, private repositories etcetera. You can pick specific tags and branches to checkout.

For managing the sources it is recommended to create a *sources.cfg* which you can include in your buildout.

```
[buildout]
extends = http://plonesource.org/sources.cfg
extensions = mr.developer

auto-checkout =
    Products.PloneFormGen
    collective.supercool
```

Adding add-on package names to the **auto-checkout** list will make buildout check out the source to the *src* directory upon next buildout run.

Note: It is not recommended to use *auto-checkout* = *, especially when you extend from a big list of sources, such as the plonesource.org list.

Note: The *auto-checkout* option only checks out the source. It is also required to add the package to the *eggs* list for getting it installed, see above.

After creating a *sources.cfg* you need to make sure that it gets loaded by the *buildout.cfg*. This is done by adding it to the *extends* list in your *buildout.cfg*:

```
[buildout]
extends =
    base.cfg
    versions.cfg
    sources.cfg
```

As always: after modifying the buildout configuration you need to rerun buildout and restart your instance:

```
bin/buildout -N
bin/instance restart
```

Further help

More detailed instructions for installing Plone add-ons are available for dealing with legacy systems.

To ask if a particular add-on has already been updated to Plone 5, you can go to community.plone.org

Please visit the [help asking guidelines](#) and [Plone support options](#) page to find further help if these instructions are not enough. Also, contact the add-on author, as listed on Plone product page, to ask specific instructions regarding a particular add-on.

Installing security updates

Description

How to install security hotfixes.

Where to find the latest information

Every now and again, security updates (or ‘hotfixes’) are released. To keep your Plone site secure, you should install these when they come available.

Plone hotfixes are released after timely announcement from the Security Team.

To stay up to date, it is advisable when you are the administrator of a Plone site, to subscribe to the [Plone Announce mailing list](#)

This is a very low-volume list. Security updates are also announced via the plone.org website and other channels.

To see which Hotfixes should be applied against which Plone version, you can check [the security matrix at plone.org/security/hotfixes](https://plone.org/security/hotfixes)

When a new maintenance release is brought out, the previous Hotfixes are incorporated. If you run Plone 4.3.3 or later, there is no need to install hotfixes for the Plone 4.x series, and the Plone 5 release has all relevant previous updates as well. (state: September 2015. Do check the link above if reading at a later date)

How to install

Each security hotfix comes with installation instructions, but in general it is no different than installing *any other add-on*

Guide to deploying and installing Plone in production

Description

A guide to the Plone deployment stack, including load balancing, proxy caching, server preparation, backups, log rotation, and process control.

This guide particularly focuses on [Unix-like](#) environments, though the stack discussion may be useful to everyone.

Introduction

The purpose of this guide; its audience and assumptions

This guide is an overview of how to set up Plone and its supporting software stack for production purposes on one of the popular Unix work-alike operating systems.

We'll cover platform preparation and basic considerations for installation of Plone itself. We'll also go into common setups of the other parts of the deployment stack needed for real-life deployment:

- A general-purpose web server to handle URL rewriting and integration with other web components;
- Load balancing;
- Server-side caching;
- Backup;
- Log rotation;
- Database packing

We won't cover the details of installation or actual tool setup. Those are better covered in the [Installation](#) and other [Hosting](#) guides.

And, what about Windows?

Production deployment for Windows is typically very different from that on Unix-like systems. While some parts of the common open-source stack are available on Windows (Apache, for example), it's more common to integrate using tools like IIS that are often already in use in the enterprise. If your shop is committed to a Microsoft stack, this document won't be of much use to you. However, if you're on a Windows server, but open to using the (very often superior) open-source alternatives to Microsoft application components, the stack and tools discussion here may be very useful.

Audience

There is one audience for this document: system administrators who wish to deploy Plone for a production server. We assume that you know how to install and configure your operating system, including its package manager or port collection, file system, user permissions, firewalls, backup and logging mechanisms.

You should be able to use the command-line shell and able to translate between the file paths and hostnames used in examples and the ones you'll be using on your deployed server.

You'll need root access (or sudo privileges) adequate to install packages, create users and set up cron jobs.

The instructions below have been tested with clean OS platforms created on commodity cloud servers.

Assumptions

We'll be describing base-level production deployments that will meet many, but not all needs. And we'll be using the most commonly used and widely supported tools for the stack. Tools like Apache, Nginx, haproxy and rsync.

You may have other needs (like integration with LDAP or a relational database) or wish to use other tools (Apache Traffic Server, Varnish, squid ...). That's fine, and there are many good documents elsewhere in the plone.org documentation section that cover these needs and tools.

Security considerations

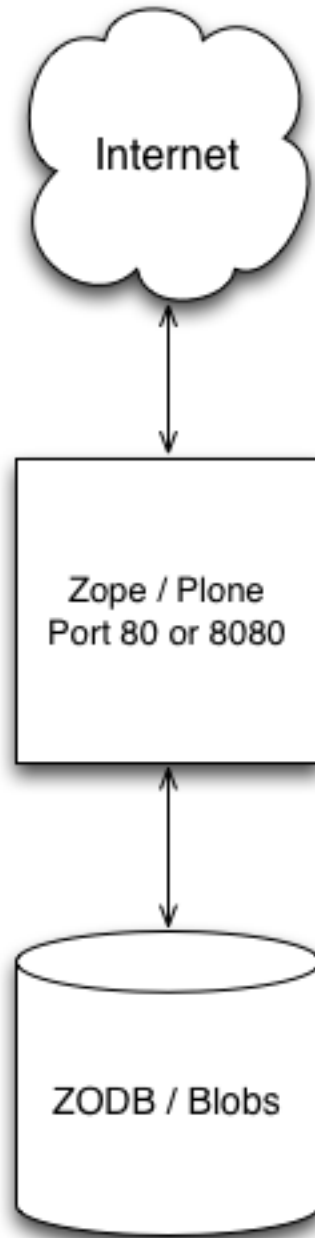
The approaches we describe here are practical for many Internet and Intranet servers. However, they should be considered a baseline and may not meet your security needs. Plone can be deployed in chroot jails or OpenSolaris zones or with much more compartmentalized process and file ownership if your application requires a greater degree of protection.

Background: the stack

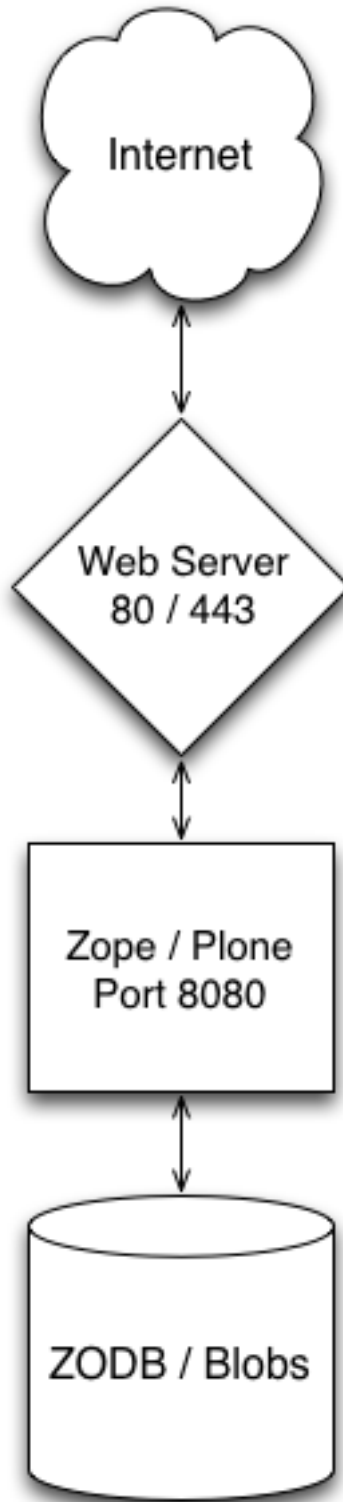
Many integrators arrive at Plone after previously working with PHP applications. They are used to using Apache with modPHP and an SQL server. This common application-server software stack is often deployed from the start on a pre-configured server, and installation of a PHP application may require little more than unpacking a set of files to a particular point in the file system.

The Plone application is a different animal. It runs on its own application server (Zope), and in common installations uses the Zope Object Database (ZODB) rather than an SQL database. It is nearly always deployed in conjunction with other tools, like web servers (e.g., Apache or Nginx), load balancers (like haproxy), and reverse-proxy caches (like varnish or squid). This is what we'll be discussing in this guide.

It is actually possible to deploy Plone/Zope as a stand-alone web server. If you do a simple "standalone" installation of Plone, you will end up with a working web server answering requests on port 8080 (which may be changed).



While there are production servers doing just this (typically changing the listening port from 8080 to 80), it's uncommon. It's much more common to put a general-purpose web server like Apache or Nginx between Plone and the Internet:



In this stack, it's the general-purpose web server that is connected to well-known Internet ports. Apache or Nginx answers those request and forwards them to Plone. It does so by proxying the requests.

If Plone/Zope has a built-in web server, why do you need another?

- The bare Zope server is fine for development, but does not have the strong security and forensic mechanisms of HTTP servers like Nginx or Apache. **Zope/Plone is not meant to be exposed to the Internet without a guarding reverse-proxy server.**
- You may wish to use Zope and Plone as part of a hybrid system with other best of breed components providing parts of your web site. For example, Plone is not really meant for the kind of database applications that require a relational solution. A good, general-purpose web server like Apache or Nginx serves as a great mechanism for dispatching different requests to different, best-of-breed components. They're also great for quickly serving static resources.
- Even in the simplest installation, it's desirable to do some URL rewriting to map URLs to data in different ways. This is nearly mandatory when building a hybrid system.
- Plone does not have built-in SSL support. A general-purpose web server will have a hardened SSL layer and a mechanism for handling certificates.
- You may wish to solve authentication and logging problems at a shared, higher level.

Load balancing

The deployment above may meet your needs for light-traffic sites. Its principle limitation is that it will make use of only a single processor and file system to render Plone pages.

The Zope application server allows us to divide the chores of rendering web pages (very CPU-intensive) from those of maintaining the file-system database. Further, we may have as many page rendering clients as we wish, all using a single database server.

The components of this mechanism, Zope Enterprise Objects (ZEO) are:

ZEO Clients Web servers in themselves, which answer requests for pages, gather page component objects from the database server, render pages and return them to the requestor.

ZEO Server Handles read/write requests for the object database from ZEO Clients. Not HTTP servers, and not meant to be visible to the Internet.

It is typical in a high-demand server situation to deploy as many ZEO clients as you have CPU cores available. More is not useful (except for a spare, debug client). ZEO clients are generally CPU/RAM-intensive. The ZEO server is a heavy disk-system user.

For multiple ZEO clients to be actually useful, you need a load-balancing front-end to distribute requests among the clients. The load balancer receives http requests and proxies them among a pool of ZEO clients.

Apache and Nginx have built-in load-management capabilities, which can allow you to combine those two layers of the stack. A dedicated load balancer like [haproxy](#) offers better features for distributing load among clients and for checking and maintaining status.

How many ZEO clients, how much memory?

It's typical to allocate roughly one ZEO client for every processor core you have available. However, there are lots of trade-offs, and many clients will eat RAM rapidly. *About Instances, Threads and RAM consumption* is a good guide to the issues involved.

Sticky sessions

As a rule of thumb, you'll tend to get better performance if you can direct requests from the same browser client to the same Zope instance (ZEO client). That's because the memory cache of the ZEO-client is more likely to be loaded with information useful for rendering requests from that source. The effect can be particularly strong for logged-in users.

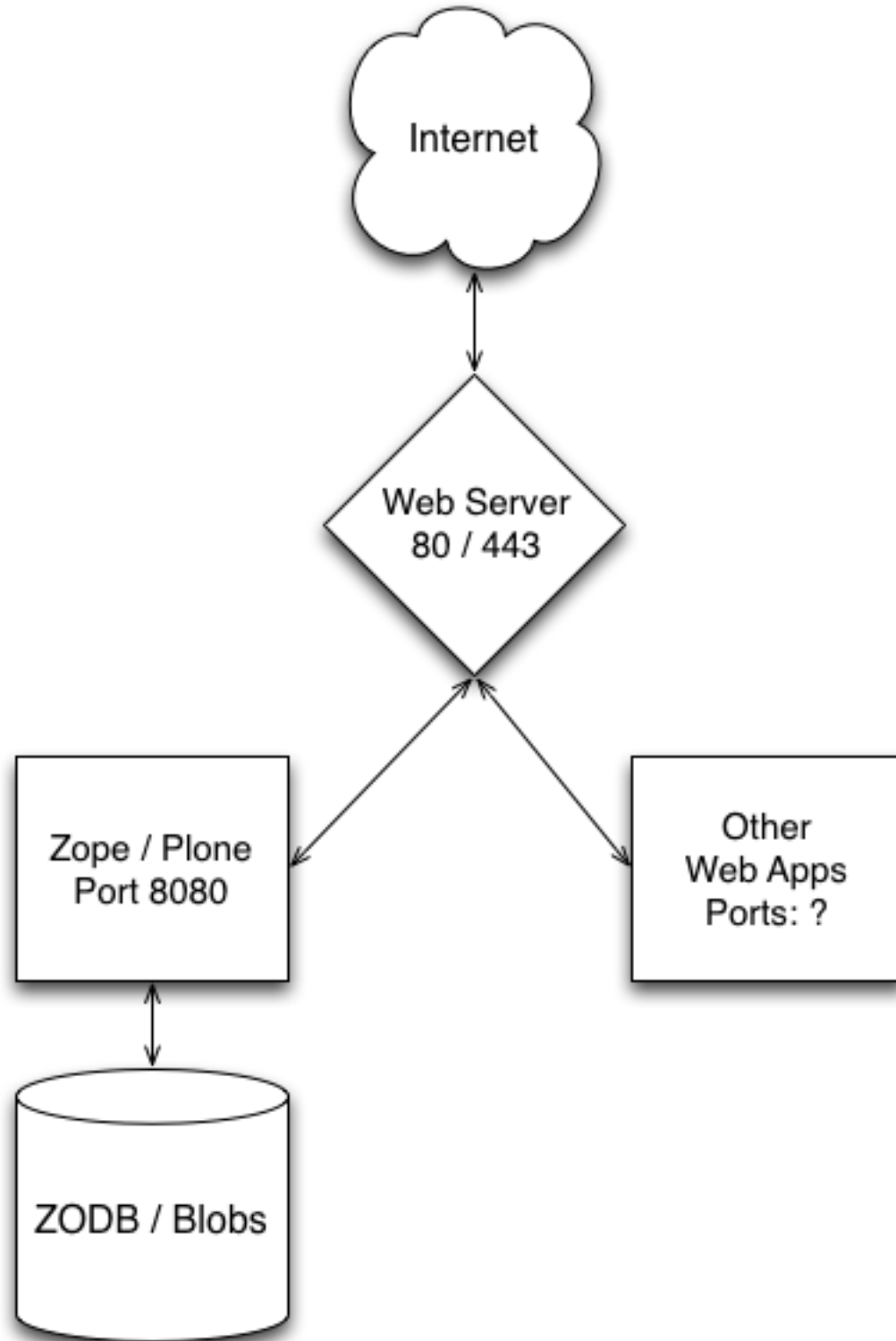


Fig. 5.1: Zope + Web Server + Web Apps

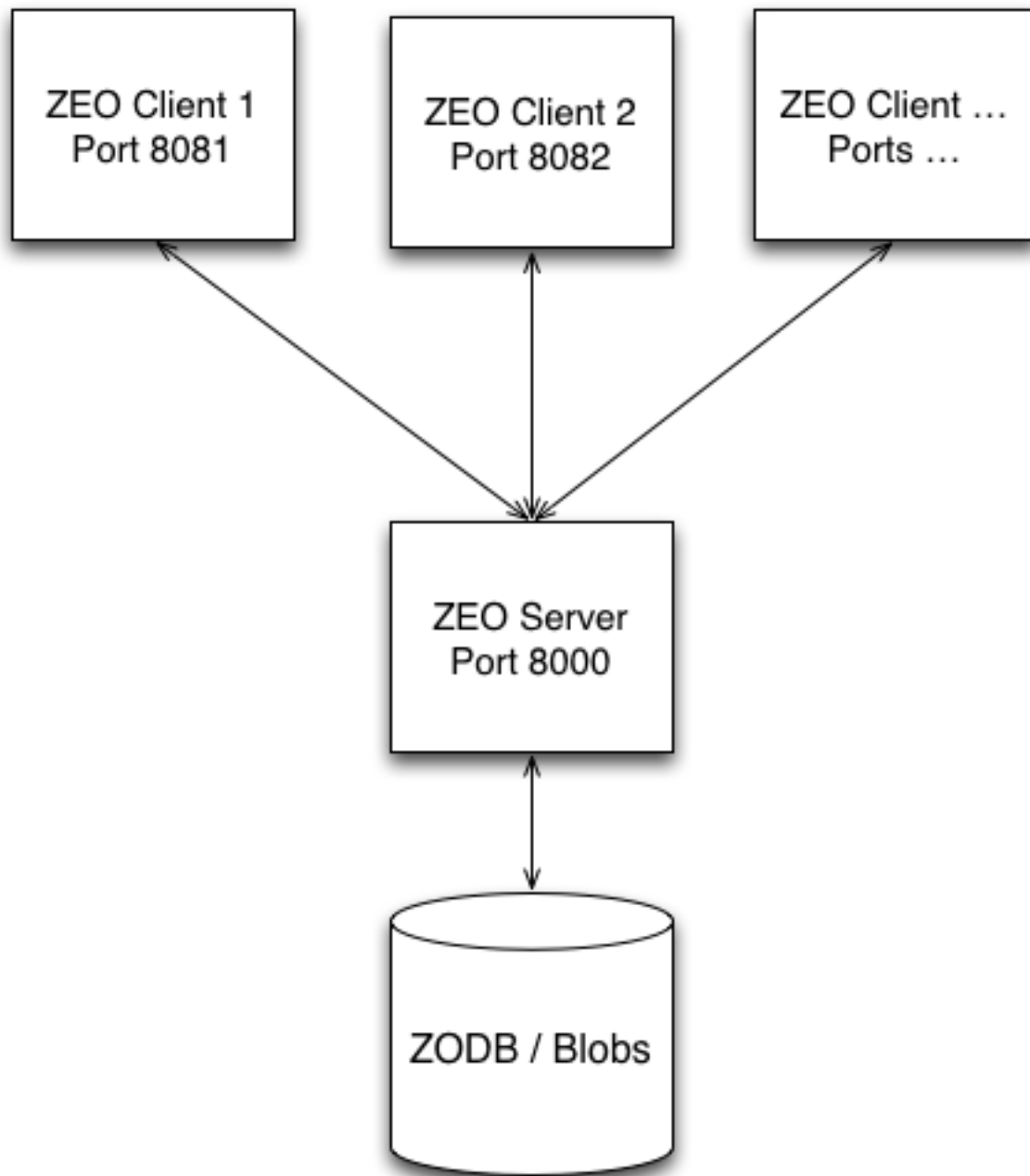


Fig. 5.2: ZEO Cluster

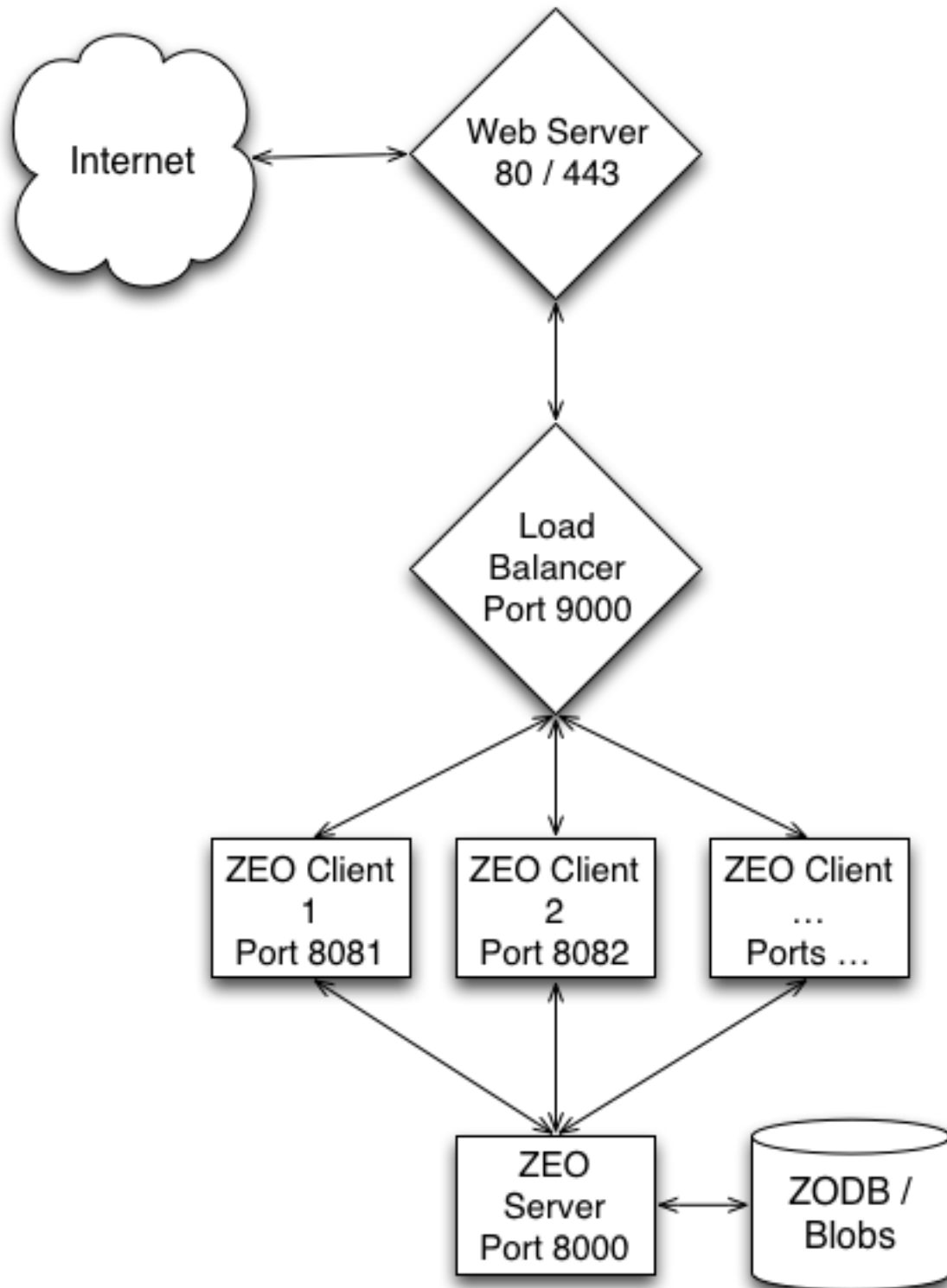


Fig. 5.3: ZEO Cluster with Load Balancer

This is not a firm rule, though. The more memory allocated to ZEO client caches, the smaller the effect. Also, if a large portion of your traffic is from search engines, benefits will be negligible. (In extreme cases, it may even be desirable to direct all your search engine traffic to the same ZEO client so that their atypical behavior doesn't spoil limited cache memory with infrequently requested pages.)

Most load balancers have some sort of mechanism for causing traffic from a single source to stick to a single ZEO client. The simplest schemes use IP addresses; cookies may also be used.

Connections and queues

A versatile load balancer like haproxy will give you fine-grained control over the queuing of connections to ZEO client back-end servers. A factor to take into account is that ZEO clients will always accept connections - even when all threads are busy. Given that requests take dramatically different amount of times to render and return, this may result in some clients having queued connections when other clients are free.

The general solution for this is to set the maximum connections per back-end ZEO client to roughly the number of threads they serve. For haproxy, this is the `maxconn` setting for the `listen` directive. This is only a rule of thumb. ZEO clients actually spawn threads as needed to return blobs, and are very efficient at that.

How severely you limit connections per client should depend on your balance of page to blob serves.

Server-side HTTP caching

When a web browser requests and receives a web resource, it silently saves the page or graphic in a local disk cache in case it may be needed later. This saves the expense of downloading the resource again.

A server-side HTTP cache does much the same thing. After Plone renders a resource, which may be a very expensive process, it saves the rendered resource in case it should be requested again soon.

With a caching reverse proxy in place, our delivery stack looks like:

As with load balancing, Apache and Nginx have built-in proxy caching abilities. If those are inadequate, Varnish or Squid are often used.

Nothing is simple about caching. There is always a trade off between currency of delivered data and the performance of the cache layer. Cache tuning for truly busy sites requires extensive measurement and experimentation, often with business decisions about the expense of currency loss vs enhanced servers.

Caching setup in Plone

In their basic outlines, browser and server-side caching work the same way. The browser or the server caches resources against the possibility that they may be needed again soon. But, how does the caching agent make the decision about how long to store a resource? Or, if it should be stored at all?

Generally, these decisions are made on the basis of caching hints that are in the HTTP response headers. The web server, or Plone, may indicate that a resource should not be cached at all, that it may be safely cached for a certain period of time, or that the caching agent should revalidate the resource by asking the server if it's changed.

Out of the box, Plone is very conservative. It assumes that currency is critical, and only tells the caching agent to store resources like icons. You may tune this up considerably by installing the *HTTP Caching* add on via the Plone control panel, then using the `* caching*` configlet to set cache rules.

The *HTTP Caching* add on is shipped with Plone, but not activated. You don't need to add it to your buildout packages. Just activate it and go. By the way, the package that does the work is `plone.app.caching`, and that's how it's often discussed.

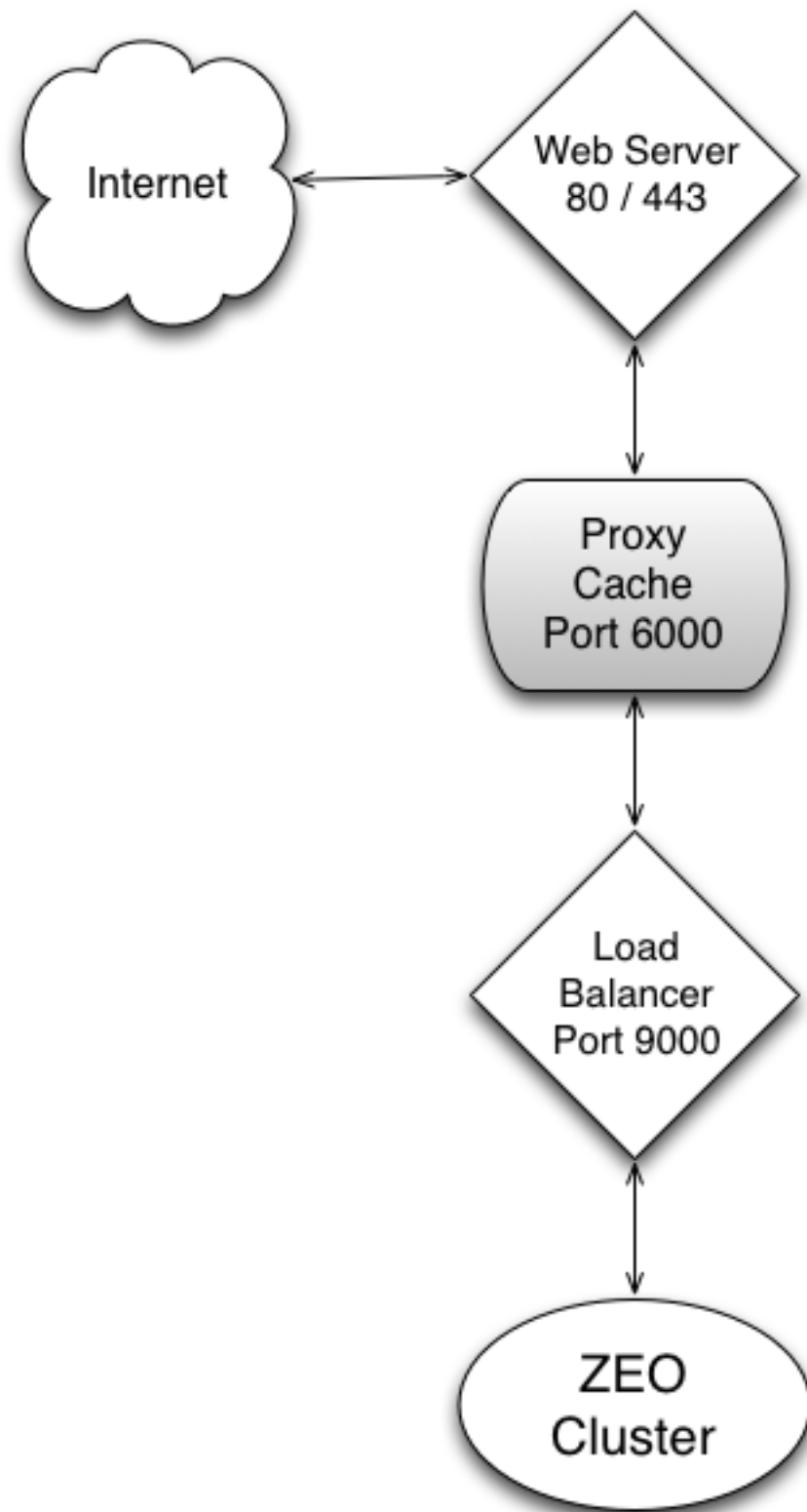


Fig. 5.4: ZEO Cluster with Server-Side Caching

Caching settings

Import settings

This is actually the place to start if you're new to cache settings. Set a basic profile by making a choice from this menu. Then use `Change settings` to refine it to your needs.

Global settings:

Enable caching Turn this on, and you'll get some immediate improvement in cache efficacy – including browser caches. Tune it up for your particular needs with the other panes in this configuration panel.

Caching Proxies

Think of this as the Varnish/Squid settings page, as it's mainly concerned with cache purging, which is typically not supported by web server proxy caches.

Cache purging is when an application server sends a message to a proxy cache to tell it that a resource needs refreshing. Cache purging is generally desirable when you're using more aggressive caching rules. If you are not setting rules to cache pages and other dynamic content, you don't need to worry about cache purging.

Caching dynamic resources like pages and trying to purge them on change is the dark, difficult side of caching. It's safest for items like files and images; hardest for the kind of complex, composite pages that are Plone's specialty.

Turn purging off and avoid aggressive caching unless you're prepared to monitor, experiment and measure.

Purge settings are extremely dependent on your proxy cache setup.

In-memory cache

Here you are offered a simple tradeoff. Memory for speed. Allocate more memory to the in-memory cache and pages are served faster. Allocate less and they're rendered more frequently. Just don't cache so much that your server starts using swap space for Zope processes.

Caching operations / Detailed settings

This is where you attach caching rules to resource types, and refine the caching rules. In general, stick with one of the profile settings (from **Import settings**) unless you're prepared to immerse yourself in caching detail.

Preparing to install Plone

Plone and Zope are generally not available via platform package or port systems.

You can't "apt-get install plone" to add it to a Debian server. (There are packages and ports out there, if you search hard enough to find them. But don't do it: they've generally had a poor record for maintenance.)

This means that you typically need to build Plone (compiling source code to binary components) on your target server. Binary installers for Plone are available for Windows and OS X, but not for Linux and BSD systems.

The OS X binary installer isn't meant for production use - though it's great for theme and add-on development and testing.

A build environment for Plone requires two sets of components:

- The GNU compiler kit and supporting components
- The development versions of system libraries required by Plone. The libraries themselves are in common use, and often included in standard distributions. But, we need the development header files.

It's generally best to install as many of these components as you can via platform packages or ports. That way, you'll be able to use your platform's automated mechanisms to keep these up-to-date, particularly with security fixes.

System python?

Plone's Unified Installer will install a suitable Python for you. However, you may wish to use your system's Python if it meets Plone's requirements. Plone 5 requires Python 2.7.

If you choose to use the system Python, you'll want to use `virtualenv` to create a virtual Python environment to isolate the Zope/Plone install from system Python packages. The Unified Installer will automatically do this for you. If you're not using the Unified Installer, learn to use `virtualenv`.

Basic build components

All installs will require the basic GNU build and archive tools: `gcc`, `g++`, `gmake`, `gnu tar`, `gunzip`, `bunzip2` and `patch`.

On Debian/Ubuntu systems, this requirement will be taken care of by installing `build-essential`. On RPM systems (RedHat, Fedora, CentOS), you'll need the `gcc-c++` (installs most everything needed as a dependency) and `patch` RPMs.

On Arch Linux you'll need `base-devel` (installs most everything needed as a dependency).

System Python

If you're using your system's Python, you will need to install the Python development headers so that you'll be able to build new Python components. On Debian/Ubuntu systems, this is usually the `python-dev` package. Port installs will automatically include the required `python.h` requirement as part of their build process.

If you're using your system Python, you will not need the `readline` and `libssl` development packages mentioned below. The required libraries should already be linked to your Python.

System libraries

For any install, the development versions of:

- `libssl`
- `libz`
- `libjpeg`
- `readline`
- `libxml2/libxslt`

If you're using the System Python, add:

- `build-essential` (`gcc/make` tools)
- `python-dev`

Without the system Python (Unified Installer builds Python):

`build-essential` (`gcc/make`)

You may also need to install dependencies needed by [Pillow](https://pillow.readthedocs.org/en/latest/installation.html) a fork of the Python Image Library. For further information please read: <https://pillow.readthedocs.org/en/latest/installation.html>

Optional libraries

If Plone can find utilities that convert various document formats to text, it will include them in the site index. To get PDFs and common office automation formats indexed, add:

- poppler-utils (PDFs)
- wv (office docs)

Development versions are not needed for these.

Platform notes

Debian/Ubuntu

Note: If you want to use System Python Packages with Ubuntu 16.04 you need to install them:

- python2.7
- python2.7-dev

```
apt install python2.7 python2.7-dev
```

Use `apt install`. The matching package names are:

- build-essential
- libssl-dev
- libz-dev
- libjpeg-dev
- libreadline-dev
- libxml2-dev
- libxslt-dev
- python-dev

Fedora

Using `dnf install`:

```
gcc-c++ patch openssl-devel libjpeg-devel libxslt-devel readline-devel make which python-devel wv poppler-utils
```

CentOS

Using `yum install`:

```
gcc-c++ patch openssl-devel libjpeg-devel libxslt-devel readline-devel make which python-devel wv poppler-utils
```

OpenSUSE

Using `zypper` in

- `gcc-c++`
- `make`
- `readline-devel`
- `libjpeg-devel`
- `zlib-devel`
- `patch`
- `libopenssl-devel`
- `libexpat-devel`
- `man`

`--build-python` will be needed as the system Python 2.7 is missing many standard modules.

Arch Linux

Using `pacman -S`

- `base-devel`
- `libxml2`
- `libxslt`
- `libjpeg-turbo`
- `openssl`

OS X

Installing XCode and activating the optional command-line utilities will give you the basic GNU tools environment you need to install Plone with the Unified Installer. You may also use MacPorts (the BSD ports mechanism, tailored to OS X) to install `libjpeg`, `libxslt` and `readline`. If you do, remember to keep your ports up-to-date, as Apple's updates won't do it for you.

Creating a Plone user

While testing or developing for Plone, you may have used an installation in a home directory, owned by yourself. That is not suitable for a production environment. Plone's security record is generally excellent, however there have been - and probably will be again in the future - vulnerabilities that allow an attacker to execute arbitrary commands with the privileges of the process owner. To reduce this kind of risk, Plone - and all other processes that allow Internet connections - should be run with user identities that have the minimum privileges necessary to maintain their data and write logs.

In a Unix-workalike environment, the most common way of accomplishing this is to create a special user identity under which you will run Plone/Zope. That user identity should ideally have no shell, no login rights, and write permissions adequate only to change files in its `/var` directory.

The ideal is hard to achieve, but it's a good start to create an unprivileged “plone” user, then use “sudo -u plone command” to install Plone and run buildout. This is basically what the Unified Installer will do for you if you run its install program via sudo. The installer uses root privileges to create a “plone” user (if one doesn't exist), then drops them before running buildout.

Don't run buildout as root!

Don't use bare “sudo” or a root login to run buildout. Buildout fetches components from the Python Package Index and other repositories. As part of package installation, it necessarily executes code in the setup.py file of each package.

Starting, stopping and restarting

If you're using a stand-alone Zope/Plone installation (not a ZEO cluster), starting and stopping Plone is easy. A production ZEO cluster deployment adds some complexity because you'll now be controlling several process: a ZEO server and several ZEO clients.

If you check the “bin” directory of your buildout after building a cluster, you'll find control commands for the server and each client. They're typically named zeoserver, client1, client2, client#. You can do a quick start with the command sequence:

```
cd /var/db/your_plone_build
sudo -u plone_daemon bin/zeoserver start
sudo -u plone_daemon bin/client1 start
sudo -u plone_daemon bin/client2 start
...
```

If you've set all this up with the Unified Installer, you'll have a convenience controller script named “plonectl” that will start all your components with one command:

```
cd /var/db/your_plone_build
sudo -u plone_daemon bin/plonectl start
```

Each “start” command will run the program in “daemon” mode: after a few startup messages, the program will disconnect from the console and run in the background.

The daemon mode start will write a process ID (pid) file in your buildout's “var” directory; that pid may be used to control the background process. It's automatically used by “stop” and “restart” commands.

Starting and stopping Plone with the server

You can start and stop Plone with your server by adding an init.d (Linux and other sys v heritage systems) or rc.d (BSD heritage) script that accepts start and stop commands.

The Unified Installer has an init_scripts directory that contains sample initialization/stop scripts for several platforms. If you didn't use that installer, you may find the scripts on [github](#).

Process control with Supervisor

A much better alternative to custom init scripts is to use a process-control system like [Supervisor](#). Supervisor is well-known by the Plone community, and you should have no trouble getting community support for it. It's available as a package or port on most Linux and BSD systems (look for supervisor, supervisord or supervisor-python). Installing the port or package will typically activate supervisor. You then just add the Zope/Plone commands to the supervisor configuration file.

Process-control systems like supervisor typically require the controlled application to run in foreground or console mode. Don't confuse this with the Zope/Plone "fg" command, which runs the application in debug mode (which is *very* slow). Instead, use "console" for clients. Use "fg" for the zeoserver; it doesn't have the "console" command, but its performance is unhindered.

Here's a sample program-configuration stanza for supervisor, controlling both a ZEO server and client:

```
[program:plone_zeoserver]
command=/var/db/plone/zeocluster/bin/zeoserver fg
user=plone_daemon
directory=/var/db/plone/zeocluster
stopwaitsecs=60

[program:plone41_client1]
command=/var/db/plone41/zeocluster/bin/client1 console
user=plone_daemon
directory=/var/db/plone41/zeocluster
stopwaitsecs=60
```

Note the "stopwaitsecs" setting. When trying to stop a program, supervisor will ordinarily wait 10 seconds before trying aggressive measures to terminate the process. Since it's entirely possible for a ZEO client to take longer than this to stop gracefully, we increase the grace period.

When running a ZEO cluster through a process-control system such as supervisor, you should always use the system's own control mechanisms (supervisorctl for supervisor) to start, stop, and status-check cluster components.

Cluster restarts

Using multiple ZEO clients and a load balancer makes it possible to eliminate downtime due to ZEO client restarts. There are many reasons why you might need to restart clients, the most common being that you have added or updated an add-on product. (You should, of course, have tested the new or updated package on a staging server.)

The basic procedure is simple: just restart your clients one at a time with a pause between each restart. This is usually scripted.

Load balancers, however, may raise issues. If your load balancer does not automatically handle temporary node downtime, you'll need to add to your client restart recipe a mechanism to mark clients as in down or maintenance mode, then mark them "up" again after a delay.

If your load balancer does handle client downtime, you may still need to make sure that it doesn't decide the client is "up" too early. Zope instances have a "fast listen" mode that causes them to accept HTTP requests very early in the startup process – many seconds before they can actually furnish a response. This may lead your load balancer to diagnose the client as "up" and include it in the cluster. This can lead to some very slow responses. To improve the situation, turn off the "fast listen" mode in your client setup:

```
[client1]
recipe = plone.recipe.zope2instance
...
http-fast-listen = off
...
```

If you are unable to tolerate slow responses during restarts, even this may not be good enough. Even after a Zope client is able to respond to requests, its first few page renderings will be slow while client database caches are primed. When speed sensitivity is this important, you'll want to add to your restart script a command-line request (via wget or curl) for a few sample pages. Do this after client restart and before marking the client "up" in the cluster. This is not commonly required.

Logs and log rotation

Plone and Zope maintain a variety of log files. As with all log files, you need to rotate your logs or your server will die from lack of storage. Log rotation is a process of maintaining a set of historical log files while periodically starting the current log file anew.

Log types and locations

The buildout recipes that set up ZEO server and client components allow you to set the names and location of your log files. We'll describe below the common names and locations. If this doesn't match your situation, check your buildout's `zeoserver` and `zope2instance` sections.

Note: If Zope instance is started in the foreground mode logs will be printed in the console (stdout).

ZEO server log

A ZEO server only maintains one log file, which records starts, stops and client connections. Unless you are having difficulties with ZEO client connections, this file is uninformative. It also typically grows very slowly - so slowly that you may never need to rotate it.

The ZEO server log for a cluster will typically be found under your buildout directory at `var/zeoserver/zeoserver.log`.

Client logs

Client logs are much more interesting and grow more rapidly. There are two kinds of client logs, and each of your clients will maintain both:

Access logs

A record of HTTP, WebDAV, and - if it's turned on - ftp accesses to the client. This resembles traditional web-server log files. Typical location of a client's access log is `var/client#/Z2.log`.

Event logs

Startup, shutdown and error messages. Event logs need attention so that errors are quickly discovered. Typical location of a client's event log is `var/client#/event.log`.

Log levels

You may set the verbosity level of access and event logs via the `zope2instance` sections for your clients. In the context of deployments, it can be very useful for change the `loglevel` for access logs. The default verbosity level for access logs - `WARN` - creates an entry for every HTTP access. If you are recording HTTP accesses via your proxy server, you may change the access logging level to `"ERROR"` and dramatically slow the rate at which your access logs grow:

```
[client1]
recipe = plone.recipe.zope2instance
...
z2-log-level = ERROR
...
```

Don't turn down the access log level until you've had a chance to tune up your proxy cache. Seeing which requests make it through to the ZEO client is very useful information when checking caching and load balancing.

Client log rotation

The basic option here is between using the ZEO client log rotation mechanisms built into Zope and using external mechanisms - such as the log-rotation facilities available on your server.

Plone 4.2.2+

Plone 4.2.2+ allows you to set a simple size-based mechanism for client log rotation.

The mechanism actually is built into Zope 2.12+ (used in Plone 4.0+), but there was no easy way to use it in a buildout until release 4.2.5 of `plone.recipe.zope2instance`. That recipe version ships with Plone 4.2.2+. We'll describe later a not-as easy mechanism for earlier 4.x series releases of Plone.

For Plone 4.2.2+, just add configuration settings like these to your buildout's `zope2instance` sections:

```
[client1]
recipe = plone.recipe.zope2instance
...
event-log-max-size = 5 MB
event-log-old-files = 5
access-log-max-size = 20 MB
access-log-old-files = 10
```

This will maintain five generations of event logs of maximum five megabytes in size and 10 generations of 20 megabyte access logs.

For earlier versions of Plone in the 4.x series, you may use a custom log setup command to pass parameters to Zope:

```
[client1]
recipe = plone.recipe.zope2instance
...
event-log-custom =
    <logfile>
        max-size = 5mb
        old-files 5
    </logfile>
access-log-custom =
    <logfile>
        max-size = 20mb
        old-files 10
    </logfile>
```

Other log rotation mechanisms

Unix-ish systems have several log rotation mechanisms available. Two common ones are `newsyslog` and `logrotate`. Both are well-documented. The critical thing you need to know for each is how to signal Zope that a log rotation has occurred, forcing it to reopen the log file. Zope will do this if you send the client process a `USR2` signal.

For example, with `logrotate`, you can rotate a client's logs with a configuration like:

```
# rotate logs for client #2
/var/db/plone4/zeocluster/var/client2/Z2.log
/var/db/plone4/zeocluster/var/client2/event.log {
    rotate 5
    weekly
```

```
sharedscripts
postrotate
    kill -USR2 `cat /var/db/plone4/zeocluster/var/client2/client2.pid`
endscript
}
```

Error alerts

Zope can email access log error messages. As with other logging instructions, this is done with an addition to client zope2instance sections of your buildout:

```
[client1]
recipe = plone.recipe.zope2instance
...
mailinglogger =
    <mailing-logger>
        flood-level 10
        level error
        smtp-server localhost
        from errors@yourdomain.com
        to errors@yourdomain.com
        subject [My domain error] [%(hostname)s] %(line)s
    </mailing-logger>
```

For complete detail on configuration, see the [mailinglogger documentation](#).

Database packing

Packing is a vital regular maintenance procedure

The Plone database does not automatically prune deleted content. You must periodically pack the database to reclaim space.

Zope's object database does not immediately remove objects when they are deleted. Instead, they are just marked inactive. This has advantages: it supplies a knowledgeable administrator with the ability to undo transactions on an emergency basis. However, this means that the disk space consumed by your object database will grow with every transaction.

Packing the database reclaims the space previously consumed by deleted objects. You *must* periodically pack your database, or it will eventually consume all available disk space.

It also may be done while the system is live.

Setting up packing

On a development or testing installation, packing will be an infrequent need.

You may initiate a packing operation via the Management Interface.

It will allow you to set the number of days of transactions you wish to keep in the undo stack.

On a production system, you should pack the database via a `bin/zeopack` in your buildout directory.

zeopack is installed automatically by the `plone.recipe.zeoserver` recipe that generates the zeoserver (database server component).

You may set packing options for zeopack by setting attributes in the zeoserver part of your buildout:

```
[zeoserver]
recipe = plone.recipe.zeoserver
...
pack-days = 3
```

Will (after buildout is run), cause bin/zeopack to conserve three days of undo history during the pack operation.

Other options include:

`pack-gc`

Can be set to false to disable garbage collection as part of the pack. Defaults to true.

`pack-keep-old`

Can be set to false to disable the creation of *.fs.old files before the pack is run. Defaults to true.

`pack-user`

If the ZEO server uses authentication, this is the username used by the zeopack script to connect to the ZEO server.

`pack-password`

If the ZEO server uses authentication, this is the password used by the zeopack script to connect to the ZEO server.

Packing

Expect the packing operation to be time-consuming and for the time to grow on a linear basis with the size of your object database.

Disk-space

The packing operation will (unless you force this off) copy the existing database before it begins packing.

This means that a packing operation will consume up-to twice the space currently occupied by your object database. (Pre-existing .old files get overwritten.)

Regular scheduling

Database packing is typically run as an automated (cron) job.

The cron job may be set up in the system cron table, or in the Plone users.

Disk packing is an extremely disk-intensive operation. It is best to schedule it to occur when your monitoring indicates that disk usage is usually low.

Backing up your Plone deployment

Description

Strategies for backing up operating Plone installations.

A guide to determining what to back up and how to back it up and restore it safely.

Introduction

The key rules of backing up a working system are probably:

- Back up everything
- Maintain multiple generations of backup
- Test restoring your backups

Warning: This guide assumes that you are already doing this for your system as a whole, and will only cover the considerations specific to Plone. When we say we are assuming you're already doing this for the system as a whole, what we mean is that your system backup mechanisms - rsync, bacula, whatever - are already backing up the directories into which you've installed Plone.

Your buildout and buildout caches are already backed up, and you've tested the restore process.

Your remaining consideration is making sure that Plone's database files are adequately backed up and recoverable.

Objects in motion

Objects in motion tend to remain in motion. Objects that are in motion are difficult or impossible to back up accurately.

Translation: Plone is a long-lived process that is constantly changing its content database. The largest of these files, the Data.fs filestorage which contains everything except Binary Large Objects (BLOBs), is always open for writing. The BLOB storage, a potentially complex file hierarchy, is constantly changing and must be referentially synchronized to the filestorage.

This means that most system backup schemes are incapable of making useful backups of the content database while it's in use. We assume you don't want to stop your Plone site to backup, so you need to add procedures to make sure you have useful backups of Plone's data. (We assume that you know that the same thing is true of your relational database storage.)

Where's my data?

Your Plone instance installation will contain a `./var` directory (in the same directory as `buildout.cfg`) that contains the frequently changing data files for the instance. Much of what's in `./var`, though, is not your actual content database. Rather, it's log, process id, and socket files.

The directories that actually contain content data are:

Filestorage

`./var/filestorage:`

This is where Zope Object Database filestorage is maintained. Unless you've multiple storages or have changed the name, the key file is `Data.fs`. It's typically a large file and contains everything except BLOBS.

The other files in filestorage, with extensions like `.index`, `.lock`, `.old`, `.tmp` are ephemeral, and will be recreated by Zope if they're absent.

Blobstorage

`./var/blobstorage:`

This directory contains a deeply nested directory hierarchy that, in turn, contains the BLOBs of your database: PDFs, image files, office automation files and such.

The key thing to know about filestorage and blobstorage is that they are maintained synchronously. The filestorage has references to BLOBs in the blobstorage. If the two storages are not synchronized, you'll get errors.

collective.recipe.backup

`collective.recipe.backup` is a well-maintained and well-supported recipe for solving the “objects in motion” problem for a live Plone database. It makes it easy to both back up and restore the object database. The recipe is basically a sophisticated wrapper around `repozo`, a Zope database backup tool, and `rsync`, the common file synchronization tool.

Note: Big thanks to Reinout van Rees, Maurits van Rees and community helpers for creating and maintaining `collective.recipe.backup`.

If you're using any of Plone's installation kits, `collective.recipe.backup` is included in your install. If not, you may add it to your buildout by adding a `backup` part:

```
[buildout]
parts =
    ...
    backup
    ...

[backup]
recipe = collective.recipe.backup
```

There are several useful option settings for the recipe, all set by adding configuration information. All are documented on the [PyPI page](#). Perhaps the most useful is the `location` option, which sets the destination for backup files:

```
[backup]
recipe = collective.recipe.backup
location = /path/to/reliably/attached/storage/filestorage
blobbackuplocation = /path/to/reliably/attached/storage/blobstorage
```

If this is unspecified, the backup destination is the buildout `var` directory. The backup destination, though, may be any reliably attached location - including another partition, drive or network storage.

Operation

Once you've run buildout, you'll have `bin/backup` and `bin/restore` scripts in your buildout. Since all options are set via buildout, there are few command-line options, and operation is generally as simple as using the bare commands. `bin/restore` will accept a date-time argument if you're keeping multiple backups. See the docs for details.

Backup operations may be run without stopping Plone. Restore operations require that you stop Plone, then restart after the restore is complete.

`bin/backup` is commonly included in a cron table for regular operation. Make sure you test backup/restore before relying on it.

Incremental backups

`collective.recipe.backup` offers both incremental and full backup and will maintain multiple generations of backups. Tune these to meet your needs.

When incremental backup is enabled, doing a database packing operation will automatically cause the next backup to be a full backup.

If your backup continuity needs are extreme, your incremental backup may be equally extreme. There are Plone installations where incremental backups are run every few minutes.

Copying A Plone Site

Description

Quick instructions on how to create a copy of a Plone installation.

Introduction

These instructions tell you the basics of creating a duplicate of Plone site for testing or back-up

Prerequisites

- Ability to use file system manager to copy files from/to the remote server
- Ability to use the command line

Plone Site Contents

In order to copy a Plone site the following must be copied

- `buildout.cfg` - defines your site package configuration
- `src` folder - all add-ons you have developed yourself
- `var/filestorage/Data.fs` - ZODB database of your site
- `var/blobstorage` folder which contains file-like objects of ZODB database (BLOBs)

Other folders (eggs, downloads, parts) etc. are generated by buildout command and may be left empty.

Copying And Bootstrapping A Plone Site

- Create a new site in the destination using Plone installer and make sure you can log-in to the site with temporary admin account

- Copy var/filestorage/Data.fs from the old system to the new system - note that admin password is stored in Data.fs and the password given during the creation of a new site is no longer effective after Data.fs copy
- Copy blobs from the old system to the new system by copying var/blobstorage/ folder
- Copy src/ folder from the old system if you have any custom development code there
- Copy buildout.cfg and other .cfg files
- **Rerun buildout in order to automatically re-download and configure all Python packages needed to run the site**
 - python bootstrap.py to make the buildout use new local Python interpreter
 - Then bin/buildout to regenerate parts/ folder

Copying Site Data In UNIX Environment

Below are example UNIX commands to copy a Plone site data from a computer to another over SCP/SSH connection. The actual \$USERNAME and folder locations depend on your system configuration.

Note: A copy of the Plone site configuration must already exist on the target computer.

These instructions are for copying or creating a backup of your site data.

This operation can be performed on a running system - Data.fs is append only file and you will lose transactions which happened during the copying of the end of the file.

Copy Local To Remote

Run this command in your buildout Plone installation.

Copy Data.fs database:

```
scp -C -o CompressionLevel=9 var/filestorage/Data.fs plone@server.com:/srv/plone/site/  
↪var/filestorage
```

Copy BLOB Files Using Rsync

BLOB files contain file and image data uploaded to your site.

Since the file content rarely changes after upload, rsync can synchronize only changed files using -a (archive) flag.

```
rsync -av --compress-level=9 var/blobstorage plone@server.com:/srv/plone/site/var
```

WSGI

Introduction

WSGI is Python standard for hosting Python web services.

- <https://wsgi.readthedocs.org/en/latest/>
- <http://repoze.org/>

Plone and WSGI

Collection of inks about this topic, since this is a very special topic we also link to documentation about WSGI and Plone 4 for reference.

- [Plone 4 with WSGI](#)
- [Plone 5 with WSGI](#)

Zope Application Server

Description

Plone is usually run via the Zope application server. This document covers control and configuration of parts of the application server.

Introduction

This page contains instructions how to configure Zope application server.

Zope control command

The command for Zope tasks is `bin/instance` in buildout-based Plones (depending on how the part(s) for the Zope instance(s) was named in the buildout configuration file; here, it's `instance`).

List available commands:

```
bin/instance help
```

For older Plone releases, the command is `zopectl`.

If you have installed a ZEO cluster, you may have multiple instances, typically named `client1`, `client2` Substitute `client#` for `instance` below. The `zeoserver` part must be running before you may directly use a client command:

```
bin/zeoserver start
bin/client1 help
```

Adding users from command-line

In case you need to reset/recover the `admin` password/access.

Note: You cannot override an existing `admin` user, so you probably want to add `admin2`.

You need to do this when you forget the admin password or the database is damaged.

Add user with Zope Manager permissions.

```
bin/instance stop # stop the site first
bin/instance adduser $USERNAME $PASSWORD
bin/instance start
```

More info

- <https://plone.org/documentation/faq/locked-out>

Timezone

Add to the `[instance]` part in `buildout.cfg`:

```
environment-vars =
    TZ Europe/Helsinki
```

Log level

The default log level in Zope is `INFO`. This causes a lot of logging that is usually not needed.

To reduce the size of log files and improve performance, add the following to the `[instance]` part (the part(s) that specify your Zope instances) in `buildout.cfg`:

```
event-log-level = WARN
z2-log-level = CRITICAL
```

Creating additional debug instances

You might want to keep your production `buildout.cfg` and development configuration in sync automatically as possible.

A good idea is to use the same `buildout.cfg` for every Plone environment. For conditional things, such as turning debug mode on, extend the buildout sections, which in turn create scripts to launch additional Zope clients in the `bin/` folder:

```
[instance]
recipe = plone.recipe.zope2instance
zope2-location = ${zope2:location}
user = admin:x
http-address = 8080
debug-mode = off
verbose-security = off

...

environment-vars =
    PTS_LANGUAGES en fi

#
# Create a launcher script which will start one Zope instance in debug mode
#
[debug-instance]
# Extend the main production instance
<= instance

# Here override specific settings to make the instance run in debug mode
debug-mode = on
verbose-security = on
event-log-level = DEBUG
```

And now you can start your **development** Zope as:

```
bin/debug-instance fg
```

And your main Zope instance stays in production mode:

```
bin/instance
```

Note: Starting Zope with the `fg` command forces it into debug mode, but does not change the log level.

Virtual hosting

Zope has a component called Virtual Host Monster which does the virtual host mapping inside Zope. More information can be found in the [zope book](#)

Suppressing virtual host monster

If you ever mess up your virtual hosting rules so that Zope locks you out of the management interface, you can add `_SUPPRESS_ACCESSRULE` to the URL to disable VirtualHostMonster.

Import and export

Zope application server allows copying parts of the tree structure via import/export feature. The exported file is basically a Python pickle containing the chosen node and all child nodes.

Importable `.zexp` files must be placed on `/parts/instance/import` buildout folder on the server. If you are using clustered ZEO set-up, always run imports through a specific front-end instance by using direct port access. Note that `parts` folder structure is pruned on each buildout run.

When files are placed on the server to correct folder, the *Import/Export* tab in the Management Interface will pick them up in the selection drop down. You do not need to restart Zope.

More information

- <http://quintagroup.com/services/support/tutorials/import-export-plone/>

Regular database packing

The append-only nature of the *ZODB* makes the database grow continuously even if you only edit old information and don't add any new content. To make sure your server's hard disk does not fill up, you need to pack the ZODB automatically and regularly.

More info

- <http://stackoverflow.com/questions/5300886/what-is-the-suggested-way-to-cron-automate-zodb-packs-for-a-production-plone-i>

Copying a remote site database

Below is a UNIX shell script to copy a remote Plone site(s) database to your local computer. This is useful for synchronizing the development copy of a site from a live server.

```
copy-plone-site.sh
```



```
#!/bin/sh
#
# Copies a Plone site data from a remote computer to a local computer
#
# Copies
#
# - Data.fs
#
# - blobstorage
#
# Standard var/ folder structure is assumed in the destination
# and the source
#

if [ $# -ne 2 ] ; then
cat <<EOF
$0
Copy a remote Plone site database to local computer over SSH
Error in $0 - Invalid Argument Count
Syntax: $0 [SSH-source to buildout folder] [buildout target folder]
Example: ./copy-plone-site.sh yourserver.com:/srv/plone/mysite .
EOF
exit 64 # Command line usage error
fi

SOURCE=$1
TARGET=$2

STATUS=`$TARGET/bin/instance status`

if [ "$STATUS" != "daemon manager not running" ] ; then
    echo "Please stop your Plone site first"
    exit 1
fi

rsync -av --progress --compress-level=9 "$SOURCE"/var/filestorage/Data.fs "$TARGET"/
↪var/filestorage

# Copy blobstorage on rsync pass
# (We don't need compression for blobs as they most likely are compressed images_
↪already)
rsync -av --progress "$SOURCE"/var/blobstorage "$TARGET"/var/
```

Pack and copy big Data.fs

Pack Data.fs using the pbzip2, efficient multicore bzip2 compressor, before copying:

```
# Attach to a screen or create new one if not exist so that
# the packing process is not interrupted even if you lose a terminal
screen -x

# The command won't abort in the middle of the run if terminal lost
cd /srv/plone/yoursite/zeocluster/var/filestorage
tar -c --ignore-failed-read Data.fs | pbzip2 -c > /tmp/Data.fs.tar.bz2

# Alternative version using standard bzip2
# tar -c --ignore-failed-read -jf /tmp/Data.fs.tar.bz2 Data.fs
```

Then copy to your own computer:

```
scp unixuser@server.com:/tmp/Data.fs.tar.bz2 .
```

... or using `rsync` which can resume:

```
rsync -av --progress --inplace --partial user@server.com:/tmp/Data.fs.tar.bz2 .
```

Creating a sanitized data drop

A *sanitized* data drop is a Plone site where:

- all user passwords have been reset to one known one;
- all history information is deleted (packed), so that it does not contain anything sensitive;
- other possible sensitive data has been removed.

It should safe to give a sanitized copy to a third party.

Below is a sample script which will clean a Plone site in-place.

Note: Because sensitive data varies depending on your site this script is an example.

How to use:

- Create a temporary copy of your Plone site on your server, running on a different port.
- Run the cleaner by entering the URL. It is useful to run the temporary copy in foreground to follow the progress.
- Give the sanitized copy away.

This script has two options for purging data:

- *Safe purge* using the Plone API (slow, calls all event handlers).
- *Unsafe purge* by directly pruning data, rebuilding the catalog without triggering the event handlers.

The sample `clean.py`:

```
""" Pack Plone database size and clean sensitive data.
    This makes output ideal as a development drop.

    It also resets all kinds of users password to "admin".

    Limitations:

    1) Assumes only one site per Data.fs

    TODO: Remove users unless they are whitelisted.
"""

import logging
import transaction

logger = logging.getLogger("cleaner")
```

```

# Folders which entries are cleared
DELETE_POINTS = """
intranet/mydata

"""

# Save these folder entries as sample
WHITELIST = """
intranet/mydata/sample-page
"""

# All users will receive this new password
PASSWORD="123123"

def is_white_listed(path):
    """
    """
    paths = [ s.strip() for s in WHITELIST.split("\n") if s.strip() != "" ]

    if path in paths:
        return True
    return False

def purge(site):
    """
    Purge the site using standard Plone deleting mechanism (slow)
    """
    i = 0
    for dp in DELETE_POINTS.split("\n"):

        dp = dp.strip()
        if dp == "":
            continue

        folder = site.unrestrictedTraverse(dp)

        for id in folder.objectIds():
            full_path = dp + "/" + id
            if not is_white_listed(full_path):
                logger.info("Deleting path:" + full_path)
                try:
                    folder.manage_delObjects([id])
                except Exception, e:
                    # Bad delete handling code - e.g. catalog indexes b0rk out
                    logger.error("*** COULD NOT DELETE ***")
                    logger.exception(e)
                i += 1
            if i % 100 == 0:
                transaction.commit()

def purge_harder(site):
    """
    Purge using forced delete and then catalog rebuild.

    Might be faster if a lot of content.
    """
    i = 0

    logger.info("Kill it with fire")

```

```
for dp in DELETE_POINTS.split("\n"):

    if dp.strip() == "":
        continue
    folder = site.unrestrictedTraverse(dp)

    for id in folder.objectIds():
        full_path = dp + "/" + id
        if not is_white_listed(full_path):
            logger.info("Hard deleting path:" + full_path)
            folder._delObject(id, suppress_events=True)

        i += 1
        if i % 100 == 0:
            transaction.commit()

site.portal_catalog.clearFindAndRebuild()

def pack(app):
    """
    @param app Zope application server root
    """
    logger.info("Packing database")
    cpanel = app.unrestrictedTraverse('/Control_Panel')
    cpanel.manage_pack(days=0, REQUEST=None)

def change_zope_passwords(app):
    """
    """
    logger.info("Changing Zope passwords")
    # Products.PluggableAuthService.plugins.ZODBUserManager
    users = app.acl_users.users
    for id in users.listUserIds():
        users.updateUserPassword(id, PASSWORD)

def change_site_passwords(site):
    """
    """
    logger.info("Changing Plone instance passwords")
    # Products.PlonePAS.plugins.ufactory
    users = site.acl_users.source_users
    for id in users.getUserIds():
        users.doChangeUser(id, PASSWORD)

def change_membrane_password(site):
    """
    Reset membrane passwords (if membrane installed)
    """

    if not "membrane_users" in site.acl_users.objectIds():
        return

    logger.info("Changing membrane passwords")
    # Products.PlonePAS.plugins.ufactory
    users = site.acl_users.membrane_users
    for id in users.getUserNames():
        try:
```

```

        users.doChangeUser(id, PASSWORD)
    except:
        # XXX: We should actually delete membrane users before content folders
        # or we will break here
        pass

class Cleaner(object):
    """
    Clean the current Plone site for sensitive data.

    Usage::

        http://localhost:8080/site/@@create-sanitized-copy

    or::

        http://localhost:8080/site/@@create-sanitized-copy?pack=false

    """

    def __init__(self, context, request):
        self.context = context
        self.request = request

    def __call__(self):
        """
        """
        app = self.context.restrictedTraverse('/') # Zope application server root
        site = self.context.portal_url.getPortalObject()

        purge_harder(site)
        change_zope_passwords(app)
        change_site_passwords(site)
        #change_membrane_password(site)

        if self.request.form.get("pack", None) != "false":
            pack(app)

        # Obligatory Die Hard quote
        return "Yippikayee m%€%/ f/€%/€/ Remember to login again with new password."

```

Example view registration in ZCML requiring admin privileges to run the cleaner:

```

<browser:page
    for="Products.CMFCore.interfaces.ISiteRoot"
    name="create-sanitized-copy"
    class=".clean.Cleaner"
    permission="cmf.ManagePortal"
/>

```

Log rotate

Log rotation prevents log files from growing indefinitely by creating a new file for a certain timespan and dropping old files.

Basic Log rotation for buildout users

If you are using buildout and the `plone.recipe.zope2instance` ($\geq 4.2.5$) to create your zope installation, two parameters are available to enable log rotation. For example:

- `event-log-max-size = 10mb`
- `event-log-old-files = 3`

This will rotate the event log when it reaches 10mb in size. It will retain a maximum of 3 files. Similar directives are also available for the access log.

- `access-log-max-size = 100mb`
- `access-log-old-files = 10`

Using the unix tool “logrotate”

You need to rotate Zope access and error logs, plus possible front-end web server logs. The latter is usually taken care of your operating system.

To set-up log rotation for Plone:

- Install `logrotate` on the system (if you don't already have one).
- You need to know the effective UNIX user as which Plone processes run.
- Edit log rotation configuration files to include Plone log directories.
- Do a test run.

To add a log rotation configuration file for Plone add a file `/etc/logrotate.d/yoursite` as root.

Note: This recipe applies only for single-process Zope installs. If you use ZEO clustering you need to do this little bit differently.

The file contains:

```
# This is the path + selector for the log files
/srv/plone/yoursite/Plone/zinstance/var/log/instance*.log {
    daily
    missingok
    # How many days to keep logs
    # In our cases 60 days
    rotate 60
    compress
    delaycompress
    notifempty
    # File owner and permission for rotated files
    # For additional safety this can be a different
    # user so your Plone UNIX user cannot
    # delete logs
    create 640 root root

    # This signal will tell Zope to open a new file-system inode for the log file
    # so it doesn't keep reserving the old log file handle for even if the file is
    →deleted
    postrotate
        [ ! -f /srv/plone/yoursite/Plone/zinstance/var/instance.pid ] || kill -
    →USR2 `cat /srv/plone/yoursite/Plone/zinstance/var/instance.pid`
```

```
endscript
}
```

Then do a test run of logrotate, as root:

```
# -f = force rotate
# -d = debug mode
logrotate -f -d /etc/logrotate.conf
```

And if you want to see the results right away:

```
# -f = force rotate
logrotate -f /etc/logrotate.conf
```

In normal production, logrotate is added to your operating system *crontab* for daily runs automatically.

More info:

- <http://linuxers.org/howto/howto-use-logrotate-manage-log-files>
- <http://docs.zope.org/zope2/zope2book/MaintainingZope.html>
- <http://serverfault.com/questions/57993/how-to-use-wildcards-within-logrotate-configuration-files>

Log rotate and chroot

Note: In this example we are using the package ‘shroot’. Please make sure you have it installed

chroot ‘ed environments don’t usually get their own cron. In this case you can trigger the log rotate from the parent system.

Add in the parent `/etc/cron.daily/yourchrootname-logrotate`

```
#!/bin/sh
schroot -c yoursitenet -u root -r logrotate /etc/logrotate.conf
```

Log rotate generation via buildout using UNIX logrotate command

buildout.cfg:

```
[logrotate]
recipe = collective.recipe.template
input = ${buildout:directory}/templates/logrotate.conf
output = ${buildout:directory}/etc/logrotate.conf
```

templates/logrotate.conf:

```
rotate 4
weekly
create
compress
delaycompress
missingok
```

```
{buildout:directory}/var/log/instance1.log ${buildout:directory}/var/log/instance1-
↪Z2.log {
    sharedscripts
    postrotate
        /bin/kill -USR2 $(cat ${buildout:directory}/var/instance1.pid)
    endscript
}

${buildout:directory}/var/log/instance2.log ${buildout:directory}/var/log/instance2-
↪Z2.log {
    sharedscripts
    postrotate
        /bin/kill -USR2 $(cat ${buildout:directory}/var/instance2.pid)
    endscript
}
```

More info:

- <http://stackoverflow.com/a/9437677/315168>

Log rotate on Windows

Use `iw.rotatezlogs`

- <http://stackoverflow.com/a/9434150/315168>

Email notifications for errors

Please see:

- <http://stackoverflow.com/questions/5993334/error-notification-on-plone-4>

Adding multiple file storage mount points

- <https://pypi.python.org/pypi/collective.recipe.filestorage>

Performance and tuning

Tips how to optimize your Plone code for maximum performance.

Cache decorators

Description

How to use the Python decorator pattern to cache the result values of your computationally expensive method calls.

Introduction

Cache decorators are convenient methods for caching function return values.

Use them like this:

```
@cache_this_function
def my_slow_function():
    # This is run only once and all subsequent calls get value from the cache
    return
```

Warning: Cache decorators do not work with methods or functions that use generators (`yield`). The cache will end up storing an empty value.

The `plone.memoize` package offers helpful function decorators to cache return values.

See also *using memcached backend for memoizers*.

Cache result for process lifecycle

Example:

```
from plone.memoize import forever

@forever.memoize
def getFields(area, subject):
    """ Get all fields inside area / subject.

    Results is cached for process lifetime.

    @return: List of internal fields
    """
    schema = getSchema(area)
    return [field for field in schema if field["subject"] == subject]
```

Timeout caches

The `@ram.cache` decorator takes a function argument and calls it to get a value.

As long as that value is unchanged, the cached result of the decorated function is returned. This makes it easy to set a timeout cache:

```
from plone.memoize import ram
from time import time

@ram.cache(lambda *args: time() // (60 * 60))
def cached_query(self):
    # very expensive operation,
    # will not be called more than once an hour
```

`time.time()` returns the time in seconds as a floating point number. `//` is Python's integer division.

The result of `time() // (60 * 60)` only changes once an hour. `args` passed are ignored.

Caching per request

This pattern shows how to avoid recalculating the same value repeatedly during the lifecycle of an HTTP request.

Caching on BrowserViews

This is useful if the same view/utility is going to be called many times from different places during the same HTTP request.

The `plone.memoize.view` package provides necessary decorators for BrowserView-based classes.

```
from plone.memoize.view import memoize, memoize_contextless

class MyView(BrowserView):

    @memoize
    def getValue():
        """ This value is recalculated for every new BrowserView context
            per request.
        """
        return "something"

    @memoize_contextless
    def getValueNoContext():
        """ This value is recalculated for all context objects once per
            request.
        """
        return "something"
```

Caching on Archetypes accessors

If you have a custom *Archetypes accessor method*, you can avoid recalculating it during the request processing.

Example:

```
def getParsedORADataCached(self):
    """ Same as above, but does not run through JSON reader every time.
    """

    # Manually store the result on HTTP request object annotations

    # Use informative string + Archetypes unique identified as the key
    key = "parsed-ora-data-" + self.UID()

    cache = IAnnotations(self.REQUEST)
    data = cache.get(key, None)
    if data is not None:
        data = self.getParsedORAData()
        cache[key] = data

    return data
```

Caching using global HTTP request

This example uses the `five.globalrequest` package for caching. Values are stored on the thread-local `HTTPRequest` object which lasts for the transaction lifecycle:

```
from zope.globalrequest import getRequest
from zope.annotation.interfaces import IAnnotations

def _getProductList(self, type, language):
    """ Private implementation, builds list of products.
    """

    logger.info("Getting product list %s %s" % (type, language))
    ...
    return result

def getProductListCached(self, type, language):
    """ Public cached method, delegates to _getProductList.
    """

    request = getRequest()

    key = "cache-%s-%s" % (type, language)

    cache = IAnnotations(request)
    data = cache.get(key, None)
    if not data:
        data = self._getProductList(type, language)
        cache[key] = data

    return data
```

Testing memoized methods inside browser views

While testing browser views memoized methods, you could find out that calling a method multiple times inside a test could result in getting the same result over and over, no matter what the parameters are, because you have the same context and request inside the test and the result is being cached.

One approach to bypass this is to put your code logic inside a private method while memoizing a public method with the same name that only calls the private one:

```
from plone.memoize import view
from Products.Five import BrowserView

class MyView(BrowserView):

    def _my_expensive_method():
        """Code logic goes here.
        """
        return "something"

    @view.memoize
    def my_expensive_method():
        """We call the private method here and memoize the result.
        """
```

```
return self._my_expensive_method()
```

In your tests you can call the private method to avoid memoization.

Other resources

- [plone.memoize source code](#)
- [zope.app.cache source code](#)

RAM Cache

Introduction

The RAM cache is a Zope facility to create custom in-process caches.

Using memcached backend

By default, Zope uses an in-process memory cache. It is possible to replace this with `memcached`.

Advantages:

- All front-end clients share the cache.
- Cache survives over a client restart.

Memoizers

Memoize's RAM cache can be replaced with a `memcached` backend with the following snippet.

See the set-up for the <https://plone.org/> site as an example:

- <https://github.com/plone/Products.PloneOrg/blob/master/src/Products/PloneOrg/caching.py>

RAM Cache

The RAM cache is used e.g. as a rendered template cache backend.

You can add `MemcachedManager` to your Zope setup, and replace the `RamCache` instance in the Management Interface with a new instance of `MemcachedManager` (keep the id the same).

- <https://pypi.python.org/pypi/Products.MemcachedManager>

Using custom RAM cache

You want to use a custom cache if you think cache size or saturation will pose problems.

The following advanced example shows how to enhance existing content type text and description accessors by performing HTML transformations and caching the result in a custom RAM cache.

Example:

```

import logging

import lxml.html

from zope.app.cache import ram

from Products.feedfeeder.content.item import FeedFeederItem
from gomobile.xhtmlimp.transformers.xhtmlimp_safe import clean_xhtml_mp

logger = logging.getLogger("GoMobile")

logger.info("Running in feedfeeder monkey-patches")

# Cache storing transformed XHTML
xhtml_cache = ram.RAMCache()
xhtml_cache.update(maxAge=86400, maxEntries=1000)

# Dummy object to mark missing values from cache
_marker = object()

def cache(name):
    """ Special cache decorator which generates cache key based on context object and
    ↪ cache name """
    def decorator(fun):
        def replacement(context):
            key = str(context.UID()) + "." + name

            cached_value = xhtml_cache.query(key, default=_marker)
            if cached_value is _marker:
                cached_value = fun(context)
                xhtml_cache.set(cached_value, key)
            return cached_value
        return replacement
    return decorator

def flush_cache(name, context):
    """ Clear entry in RAMCache

    global_key is function specific key, key is context specific key.

    """
    key = context.UID() + "." + name
    xhtml_cache.invalidate(key)

#
# Modify existing body text and description accessors so that
# 1) HTML is cleaned
# 2) The result cleaned HTML is cached in RAM
#
# We do not persistently want to store cleaned HTML,
# since our cleaner might be b0rked and we want to
# regenerate cleaned HTML when needed.
#

# Run in monkey patching
FeedFeederItem._old_getText = FeedFeederItem.getText

```

```
FeedFeederItem._old_setText = FeedFeederItem.setText
FeedFeederItem._old_Description = FeedFeederItem.Description
FeedFeederItem._old_setDescription = FeedFeederItem.setDescription

@cache("text")
def _getText(self):
    """ Body text accessor """
    text = FeedFeederItem._old_getText(self)

    if text:
        # can be None
        clean = clean_xhtml_mp(text)
        print "Cleaned text:" + clean
        return clean

    return text

def _setText(self, value):
    FeedFeederItem._old_setText(self, value)
    flush_cache("text", self)

@cache("description")
def _Description(self):
    """ Description accessor """
    text = FeedFeederItem._old_Description(self)

    #print "Accessing description:" + str(text)

    # Remove any HTML formatting in the description
    if text:
        parsed = lxml.html.fromstring(text.decode("utf-8"))
        clean = lxml.html.tostring(parsed, encoding="utf-8", method="text").decode(
↪ "utf-8")
        #print "Cleaned decription:" + clean
        return clean

    return text

def _setDescription(self, value):
    FeedFeederItem._old_setDescription(self, value)
    flush_cache("description", self)

FeedFeederItem.getText = _getText
FeedFeederItem.setText = _setText
FeedFeederItem.Description = _Description
FeedFeederItem.setDescription = _setDescription
```

ZCacheable

ZCacheable is an ancient Zope design pattern for caching. It allows persistent objects that are subclasses of OFS.Cacheable to have the cache backend configured externally.

The cache type (cache id) in use is stored *persistently* per cache user object, but the cache can be created at runtime (RAM cache) or externally (memcached) depending on the situation.

Note: Do not use `ZCacheable` in new code.

It takes optional backends which must be explicitly set:

```
def enableCaching():
    pas=getPAS()
    if pas.ZCacheable_getManager() is None:
        pas.ZCacheable_setManagerId(manager_id="RAMCache")
    getLDAPPlugin().ZCacheable_setManagerId(manager_id="RAMCache")
```

The `RAMCache` above is per thread. Clearing this cache for all ZEO clients is hard.

Some hints:

It is enabled per persistent object:

```
>>> app.test2.acl_users.ZCacheable_isCachingEnabled()
<Products.StandardCacheManagers.RAMCacheManager.RAMCache instance at 0x10a064cf8>

>>> app.test2.acl_users.ZCacheable_enabled()
1
```

Get known cache backends:

```
>>> app.test2.acl_users.ZCacheable_getManagerIds()
({'id': 'caching_policy_manager', 'title': ''}, {'id': 'HTTPCache', 'title': ''}, {'id': 'RAMCache', 'title': ''}, {'id': 'ResourceRegistryCache', 'title': 'Cache for saved ResourceRegistry files'})
```

Disabling it (persistent change):

```
>>> app.test2.acl_users.ZCacheable_setManagerId(None)
>>> app.test2.acl_users.ZCacheable_enabled()
1
>>> app.test2.acl_users.ZCacheable_getManagerIds()
({'id': 'caching_policy_manager', 'title': ''}, {'id': 'HTTPCache', 'title': ''}, {'id': 'RAMCache', 'title': ''}, {'id': 'ResourceRegistryCache', 'title': 'Cache for saved ResourceRegistry files'})
>>> app.test2.acl_users.ZCacheable_isCachingEnabled()
>>> app.test2.acl_users.ZCacheable_setEnabled(False)
```

More info:

- <https://github.com/zopefoundation/Zope/blob/master/src/OFS/Cache.py>
- <https://github.com/plone/plone.app.lldap/blob/master/plone/app/ldap/ploneldap/util.py>

Other resources

- [plone.memoize source code](#).
- [zope.app.cache source code](#)

About Instances and Threads, Performance and RAM consumption

Description

Understanding how instances-per-core, threads-per-instance and ZODB-caches are influencing performance.

Introduction

In a usual production Zope/Plone setup there are some tunings possible. For a certain size of site, you need more than one Zope-instance and use [HAproxy](#) or [Pound](#) to load-balance between them. Then you may ask yourself: How many instances do I need? Next you see there is value “threads per instance” and wonder about the different recommendations: Only one thread or two, or four? And how does it effect memory usage?

Rule Of The Thumb

A good **rule-of-the-thumb** for a common setup was and still is: **two instances per core, two threads per instance, adjust the number of objects in the ZODB cache to a number that your memory is used.**

But attention! If your setup gets more complex, if you have several logged in users or only anonymous users, if you use official, fancy, specific or home-grown add-ons: This rule may not apply.

In this case you need to figure out yourself. It’s more important to understand the mechanism behind than sticking to a rule.

With recent, faster hardware and the (sometimes odd) behavior of virtual machines (which can be very very different dependent on the kind of VM) this needs slight or major adjustment.

Theory

Threads: A Zope instance is running a pool of threads. It queues an incoming request and dispatches it to a free thread. If no thread is free the request remains in the queue and is dispatched when a thread was freed. If all threads are used by long-running request-to-response cycles this may block such simple tasks as publishing a tiny icon.

Database-Connection-Pool and its Cache: Once a thread runs, it requests a ZODB database connection from the connection pool. It locks the connection so no other thread can use it. The connection pool opens a new connection if all existing connections are already in use. If the request-to-response cycle is finished and the thread is freed the connection is released back to the pool.

Memory Cache: Each connection has its own memory cache. The file-system cache is shared by all connections. Each cache can have the configured number of objects in memory. Having them in memory is important, because they are unpickled if loaded from the DB - and the process of unpickling is still expensive.

An instance may never get enough load so that all available threads are used concurrently. In this case you may find in the Management Interface (Zope-root -> Control_Panel -> Database -> Main DB) that there are only 2 connections, but you have 4 threads. That’s because there were never 4 connections used in parallel.

Instances and memory: An instance creates only a minimal memory usage overhead. If you have two instances with each 2 threads or one with 4 threads and all threads are used in both cases it wont make much a difference (~15-20MB overhead per instance at time of writing).

Now while Plone is running for some time another significant (but compared to ZODB cache low) amount of consumed RAM is used for RAM-caching inside Zope (i.e. with plone.memoize). RAM-cache is shared by all threads but not between instances. To optimize ram-caching in a multi-instance environment “memcached” may be used to optimize memory cache and cache-usage and reduce an instances memory footprint.

But anyway, most memory is used (in a common setup) to cache the ZODB.

Python GIL - global interpreter lock: Well yes, the GIL is mentioned here. In a threaded environment such as Zope is it has an impact on performance. But it is low and python was optimized over the years, also Zope has a lot of I/O which reduces the GIL impact. A good and important optimization is to set the right check interval for your machine. With `jarn.checkinterval` there's a good and simple to use tool to test for the right value.

Practice

All theory is gray. But what does this mean for your setup if the rule-of-thumb above does not apply?

Get measurements! First of all you need to check yourself what happens on your machine(s), go and learn how to use [Munin](#) (with [munin.zope](#)), HAproxy (or Pound), [tool of your choice here]. After that you'll get graphs of RAM, CPU, and load and some zope related values. HAproxy or Pound may mark a node as down because all threads were blocked by long running requests, [collective.stats](#) helps here.

More instances or more threads? This question is asked often. And can not be answered without knowing more about the Plone system. We can divide it roughly into four kinds of systems:

- Only or almost logged in users,
- Only or almost only anonymous visitors,
- Mixed with many users and lots of hardware behind,
- Mixed with few users and low-budget hardware.

If you deal with logged in users there is no easy way to cache html-pages (highly recommended anyway for all static items) in a reverse proxy cache (i.e. [Varnish](#)) in front of Plone. Zope has much more work rendering pages. To render pages, objects need to be loaded from the database. Loading is expensive. If an object is already in the DB RAM cache it decreases the time to render a page significantly. So in a setup with lots of logged in users we need to take care almost all objects are loaded already. Each thread fetches a connection from the pool, each connection has its cache. If a user now requests a page it is first logged in and zope need a bunch of objects for this from the ZODB. Also other user specific information is loaded. Then user may operate in an intranet within a specific area, so these objects also need to be loaded. If we now have i.e 1 instance with 5 threads we have up to 5 pools (5 caches). All objects of interest are loaded in worst case 5 times. If there's 1 instance with 1 thread (1 cache) data is loaded only once. But if there is only one instance with one thread a browser shooting at the web-server with lots of requests at one time fills up the request queue of the instance and may time out soon. Also a second user may want to access data at the same time, but the only thread is blocked and the CPU idles. So the best is to stick users in a load-balancer (bind it to the `__ac` cookie) to 1 instance with 2 threads (also this can be adjusted dependent on your setup, test it yourself). Provide as much instances as you can (memory-consumption and cpu-usage will stop you). In such a setup usage of [memcached](#) is highly encouraged.

If you have almost all anonymous users it is much easier. You can provide less instances (here rule-of-thumb 2 per core applies in most cases) and increase threads. Too many threads are not good, because of the GIL. You need to find the number yourself, it depends much on hardware. Here - w/o memcached configured - good results can be expected, because memory cache is used efficient. Increase objects per connection cache until your memory-consumption stops you and look always at your CPU usage.

In large mixed environments with enough budget for hardware it is easy: Divide your environment in two, one for logged in users, one for anonymous - so above applies.

In smaller mixed environments with less hardware behind you need to find your own balance. A good way is configuring your load balancer to stick logged-in users to one or two distinct instances. If there are more users this is kind of tricky and may take some time to figure out a good setup. So this is the most difficult setup.

Performance Tips

Description

Tips for Plone performance tuning and making your add-on product and customizations faster.

Profiling Plone

- <https://pypi.python.org/pypi/collective.profiler/>

Optimizing ZEO and threads

For multicore systems, which basically all production systems nowadays are, you might want to optimize Python threading vs. processes. You may also tune how many Python interpreter instructions are run before doing green thread switches in the interpreter.

- <https://mail.zope.org/pipermail/zodb-dev/2010-December/013897.html>

Debugging slow threads in production

- <https://pypi.python.org/pypi/Products.LongRequestLogger>

Memcached as session storage

Storing sessions in ZEO/ZODB does not scale well, since they are very prone to raise `ConflictErrors` if there is considerable load on the system.

Memcached provides a more scalable session backend.

For more information, see [lovely.session](#) add-on product.

Input/output performance of the server

http://plope.com/Members/chrism/iostat_debugging

Summary:

```
<mcdonc> well, the example has await at about 40X svctime.. that's pretty shitty  
<mcdonc> i mean that box was useless
```

Tuning complex configurations

<http://www.lovelysystems.com/the-decathlon-of-computer-science/>

Reducing memory usage

These tips are especially critical when running Plone on low-memory virtual private server (VPS). But using the memory tips below, and some filesystem and operating system tweaks, it is also perfectly possible to run Plone on an ARM-based Android stick, or a Raspberry Pi. See <http://polyester.github.io/>

Disable extra languages

Add `PTS_LANGUAGES` to `buildout.cfg` to declare which `.po` files are loaded on the start-up:

```
[instance]
...
environment-vars =
    PTS_LANGUAGES en fi
```

Upgrade DateTime

`DateTime` 3.x and higher use significant less memory than older versions. Pinning it to 3.0.3 (4.x not tested yet) has no known side effects on all Plone 4.1.x and 4.2.x sites, but can give up to a 20-25% reduction in memory use on lower-end hardware/virtualmachines.

Large files

How to offload blob processing from Zope:

- <http://www.slideshare.net/Jazkarta/large-files-without-the-trials>

Sessions and performance

Write transactions much worse performance-wise than read transactions.

By default, every login is a write transaction. Also, Plone needs to update the logged-in user's session timestamp once in a while to keep the session active.

With a high amount of users, you may start seeing many `ConflictErrors` (read conflicts) with ZODB.

There are some tricks you can use here:

- <http://plone.293351.n2.nabble.com/the-mysterious-case-of-the-zope-sessions-that-shouldn-t-tp5731395p5731395.html>
- <https://pypi.python.org/pypi/collective.beaker/>

ZServer thread count

This specifies how many requests one ZEO front-end client (ZServer) can handle.

The default set by buildout default is 2.

Adjust it:

```
[client1]
recipe = plone.recipe.zope2instance
...
zserver-threads = 5
```

Find good value by doing performance testing for your site.

Note: Increasing thread count is useful if your Plone site does server-to-server traffic and your Plone site needs to wait for the other end, thus blocking Zope threads.

More info:

- <https://pypi.python.org/pypi/plone.recipe.zope2instance>

XSendFile

XSendFile is an enhancement over HTTP front end proxy protocol which allows offloading of file uploads and downloads to the front end web server.

More info for Plone support:

- <https://github.com/collective/collective.xsendfile>

Guide to Caching

Description

Caching strategies to improve performance.

This guide particularly focuses on [Unix-like](#) environments, though the stack discussion may be useful to everyone.

Any dynamically generated website with a non-trivial number of visitors, will benefit from [caching](#). Resources (like text, images, CSS, JavaScript) that are used for multiple visitors are stored in a way that is fast to retrieve, so that the (often complicated) back-end server doesn't have to generate those resources for every visitor.

Plone is no exception to that. Caching in Plone is a two-step process for most larger sites.

There is an add-on called `plone.app.caching` that is shipped with Plone. On its own, it will already speed up response time quite dramatically.

You have to enable it, use the default values provided, and you will have a faster site.

You can also tweak the settings to get better performance. There is always a little trade-off to be made here, so-called 'strong' caching will be faster, but it may mean that visitors get older content.

It is usually best to set up 'strong' caching for things that don't change often, like CSS and javascript files, and 'weak' caching for actual texts.

You can also 'invalidate' content automatically when you update a piece of content, so that the front-end server knows it has to get a fresh copy when you edit a piece of content.

But `plone.app.caching` works even better together with a dedicated front-end cache, a program that is specialized in doing this work. These days, the favorite and recommended program for that is called "Varnish".

Caching Rules

Description

How to program front end caching server (Varnish, Apache) to cache the content from Plone site.

Introduction

Plone caching is configured using the `plone.app.caching` add-on.

It supplies a web user interface for cache configuration and default caching rules for Plone.

Using only the web user interface, `plone.app.caching` is very flexible already.

This document mainly deals how you can combine `plone.app.caching` with your custom code.

Internally `plone.app.caching` uses `z3c.caching` which defines programming level ZCML directives to create your cache rules.

`plone.app.caching` does both:

- front end caching server support, and
- in-memory cache in Zope.

`plone.app.caching` also defines default rules for various Plone out-of-the-box content views and item. See:

- <https://github.com/plone/plone.app.caching/blob/master/plone/app/caching/caching.zcml>

The caching operations (strong, moderate, weak) are defined in Python code itself, as they have quite tricky conditions.

You can find the default operations here:

- <https://github.com/plone/plone.app.caching/blob/master/plone/app/caching/operations/default.py>

Note: You usually don't need to override the operation classes itself. `plone.app.caching` provides web UI to override parameters, like timeout, for each rule, on the *Detailed settings* tab in cache control panel (Create per-ruleset parameters link).

Setting Per-view Cache Rules

Here is an example how you can define a cache rules for your custom view class. In this example we want to cache our site front page in Varnish, because is is very complex, and wakes up a lot of ZODB objects.

The front page is programmed using `BrowserView`.

Our front page is subject to moderate changes as new content comes in, but the changes are not time critical, so we define a one hour timeout for caching the front page.

Note: Currently, setting caching rules for view classes is not supported through the web, but using ZCML or Python is the way to go.

In our case we are also using “a dummy cache” which does not provide purging through Plone — the only way to purge the front-end proxy is to do it from the Varnish control panel.

Here is our `configure.zcml` for our custom add-on `browser` package:

```
<configure
  xmlns="http://namespaces.zope.org/zope"
  xmlns:browser="http://namespaces.zope.org/browser"
  xmlns:cache="http://namespaces.zope.org/cache"
  >

  <include package="z3c.caching" file="meta.zcml" />
```

```
<!-- Let's define a ruleset which we use to cover all almost static
      pages which get heavy traffic. This will appear in Cache
      configuration of Site setup control panel. -->
<cache:rulesetType
  name="plone.homepage"
  title="Homepage"
  description="Site homepage view"
/>

<!-- We include one BrowserView class in our ruleset. This view is being
      used at the site front page. -->
<cache:ruleset
  for=".views.CoursePage"
  ruleset="plone.homepage"
/>

</configure>
```

After defining the rule and checking that the rule appears in the caching control panel, we'll:

- assign *Moderate caching* operation to *Homepage*;
- on the *Detailed settings* tab we'll use the *Create per-ruleset* command to override timeout to be 1h instead of default 24h for *Homepage*.

Warning: Do not enable the Zope RAM cache for page templates.

You will end up having some bad page HTML in Zope's internal cache and you have no idea how to clear it.

Note: If you are testing the rule on a local computer first, remember to re-do caching control panels in the production environment, as they are stored in the database.

Testing The Rule

- First, we'll test the rule on our local development computer to make sure that it loads;
- then we'll test the rule in the production environment with Varnish to see that Varnish picks up `Expires` header

Note: To test `plone.app.caching` rules you need to run the site in production mode (not in the foreground). Otherwise `plone.app.caching` is disabled.

Here is an example of using the `wget` UNIX command-line utility (discard the retrieved document and print the HTTP response headers)

```
wget --output-document=/dev/null --server-response http://localhost:8080/
```

The output looks like this:

```

huiske-imac:tmp moo$ wget --output-document=/dev/null --server-response http://
↪localhost:8080/LS/courses
--2011-08-03 15:18:27-- http://localhost:8080/LS/courses
Resolving localhost (localhost)... 127.0.0.1, ::1
Connecting to localhost (localhost)|127.0.0.1|:8080... connected.
HTTP request sent, awaiting response...
HTTP/1.0 200 OK
Server: Zope/(2.13.7, python 2.6.4, darwin) ZServer/1.1
Date: Wed, 03 Aug 2011 12:18:55 GMT
Content-Length: 42780
X-Cache-Operation: plone.app.caching.moderateCaching
Content-Language: en
Expires: Sun, 05 Aug 2001 12:18:55 GMT
Connection: Keep-Alive
Cache-Control: max-age=0, s-maxage=3600, must-revalidate
X-Cache-Rule: plone.homepage
Content-Type: text/html; charset=utf-8
Length: 42780 (42K) [text/html]

```

We see that X-Cache-Operation and X-Cache-Rule from plone.app.caching debug info are present, so we know that it is setting HTTP headers correctly, so that the front end server (Varnish) will receive the appropriate directives.

After deploying the change in the production environment, we'll check Varnish is picking up the rule.

We fetch the page twice: first run is *cold* (not yet cached), the second run should be cached

```

wget --output-document=/dev/null --server-response http://www.site.com/courses
wget --output-document=/dev/null --server-response http://www.site.com/courses

```

The output:

```

huiske-imac:tmp moo$ wget -S http://www.site.com/courses
--2011-08-03 15:39:10-- http://www.site.com/courses
Resolving www.site.com (www.site.com)... 79.125.22.172
Connecting to www.site.com (www.site.com)|79.125.22.172|:80... connected.
HTTP request sent, awaiting response...
HTTP/1.1 200 OK
Server: Zope/(2.13.7, python 2.6.5, linux2) ZServer/1.1
X-Cache-Operation: plone.app.caching.moderateCaching
Content-Language: en
Expires: Sun, 05 Aug 2001 12:34:06 GMT
Cache-Control: max-age=0, s-maxage=3600, must-revalidate
X-Cache-Rule: plone.homepage
Content-Type: text/html; charset=utf-8
Content-Length: 43466
Date: Wed, 03 Aug 2011 12:34:14 GMT
X-Varnish: 72735907 72735905
Age: 8
Via: 1.1 varnish
Connection: keep-alive
Length: 43466 (42K) [text/html]

```

We'll see that you have **two** numbers on line from Varnish:

```
X-Varnish: 72735907 72735905
```

These are Varnish internal timestamps: when the request was pulled to the cache and when it was served.

If you see only one number on subsequent requests it means that Varnish is not caching the request (because it's fetching the page from Plone every time).

If you see two numbers you know it is OK (and you can feel the speed).

More info:

- <http://stackoverflow.com/questions/6170962/plone-app-caching-for-front-page-only>

Creating A “Cache Forever” View

You might create views which generate or produce resources (images, JS, CSS) in-fly.

If you refer this views always through content unique URL you can cache the view result forever.

This can be done

- Using `blob._p_mtime`, or similar, to get the modified timestamp of the related content item. All persistent ZODB objects have `_p_mtime`
- Setting `plone.stableResource` ruleset on the view

Related ZCML

```
<configure
  xmlns="http://namespaces.zope.org/zope"
  xmlns:browser="http://namespaces.zope.org/browser"
  xmlns:cache="http://namespaces.zope.org/cache"
  >

  <include package="z3c.caching" file="meta.zcml" />
  <include package="plone.app.caching" />

  <!-- Because we generate the image URL containing image modified timestamp,
        the URL is always stable and when image changes the URL changes.
        Thus, we can use strong caching (cache URL forever)
  -->

  <cache:ruleset
    for=".views.ImagePortletImageDownload"
    ruleset="plone.stableResource"
  />

</configure>
```

Related view code:

```
from Products.Five import BrowserView

class ImagePortletImageDownload(BrowserView):
    """
    Expose image fields as downloadable BLOBS from the image portlet.

    Allow set caching rules (content caching for this view)
    """

    def __call__(self):
        """
```



```

"""
content = self.context

# Read portlet assignment pointers from the GET query
name = self.request.form.get("portletName")
portletManager = self.request.form.get("portletManager")
imageId = self.request.form.get("image")

# Resolve portlet and its image field
manager = getUtility(IPortletManager, name=portletManager, context=content)
mapping = getMultiAdapter((content, manager), IPortletAssignmentMapping)
portlet = mapping[name]
image = getattr(portlet, imageId, None)
if not image:
    # Ohops?
    return ""

# Set content type and length headers
set_headers(image, self.request.response)

# Push data to the downstream clients
return stream_data(image)

```

When we refer to the view in `` we use modified time parameter:

```

def getImageURL(self, imageDesc):
    """
    :return: The URL where the image can be downloaded from.
    """
    context = self.context.aq_inner

    params = dict(
        portletName=self.__portlet_metadata__["name"],
        portletManager=self.__portlet_metadata__["manager"],
        image=imageDesc["id"],
        modified=self.data._p_mtime
    )

    imageURL = "%s/@@image-portlet-downloader?%s" % (context.absolute_url(), urllib.
    ↪urllib.urlencode(params))

    return imageURL

```

Related ZCML registration:

```

<browser:page
    name="image-portlet-downloader"
    for="*"
    permission="zope.Public"
    class=".views.ImagePortletImageDownload"
/>

```

Varnish 4.x

Description

Varnish is a caching front-end server. This document has notes on how to use Varnish with Plone.

Introduction

This chapter contains information about using the Varnish caching proxy with Plone.

- <http://varnish-cache.org/>

To use Varnish with Plone

- Learn how to install and configure Varnish
- Add Plone virtual hosting rule to the default varnish configuration

Installation

The recommended method to install Varnish is by using your OS package manager.

Varnish is distributed in the Debian and Ubuntu package repositories.

```
$ sudo apt-get update
$ sudo apt-get install varnish
```

For installation instructions on other operating systems check: <https://www.varnish-cache.org/releases/index.html>

You can also install Varnish using Buildout. For examples check: <https://pypi.python.org/pypi/plone.recipe.varnish>

Management console

varnishadm

You can access Varnish admin console on your server by:

```
# Your system uses a secret handshake file
varnishadm -T localhost:6082 -S /etc/varnish/secret
```

(Ubuntu/Debian installation)

Telnet console

The telnet management console is available on some configurations where `varnishadm` cannot be used. The functionality is the same.

Example:

```
ssh yourhost
# Your system does not have a secret handshake file
telnet localhost 6082
```

Note: Port number depends on your Varnish settings.

More info

- <http://opensourcehacker.com/2013/02/07/varnish-shell-singleliners-reload-config-purge-cache-and-test-hits/>

Quit console

Quit command:

```
quit
```

Purging the cache

This will remove all entries from the Varnish cache:

```
varnishadm "ban req.url ~ ."
```

Or remove all entries of JPG from the Varnish cache:

```
varnishadm "ban req.url ~ .jpg"
```

Loading new VCL to live Varnish

More often than not, it is beneficial to load new configuration without bringing the cache down for maintenance. Using this method also checks the new VCL for syntax errors before activating it. Logging in to Varnish CLI requires the `varnishadm` tool, the address of the management interface, and the secret file for authentication.

See the `varnishadm` man-page for details.

Opening a new CLI connection to the Varnish console, in a buildout-based Varnish installation:

```
parts/varnish-build/bin/varnishadm -T localhost:8088
```

Port 8088 is defined in `buildout.cfg`:

```
[varnish-instance]
telnet = localhost:8088
```

Opening a new CLI connection to the Varnish console, in a system-wide Varnish installation on Ubuntu/Debian:

```
varnishadm -T localhost:6082 -S /etc/varnish/secret
```

You can dynamically load and parse a new VCL config file to memory:

```
vcl.load <name> <file>
```

For example:

```
vcl.load newconf_1 /etc/varnish/newconf.vcl
```

... or ...

```
# Ubuntu / Debian default config
vcl.load defconf1 /etc/varnish/default.vcl
```

`vcl.load` will load and compile the new configuration. Compilation will fail and report on syntax errors. Now that the new configuration has been loaded, it can be activated with:

```
vcl.use newconf_1
```

Note: Varnish remembers `<name>` in `vcl.load`, so every time you need to reload your config you need to invent a new name for `vcl.load` / `vcl.use` command pair.

See

- <http://opensourcehacker.com/2013/02/07/varnish-shell-singleliners-reload-config-purge-cache-and-test-hits/>

Logs

To see a real-time log dump (in a system-wide Varnish configuration):

```
varnishlog
```

By default, Varnish does not log to any file and keeps the log only in memory. If you want to extract Apache-like logs from varnish, you need to use the `varnishncsa` utility.

Stats

Check live “top-like” Varnish statistics:

```
parts/varnish-build/bin/varnishstat
```

Use the admin console to print stats for you:

Uptime mgt:	8+00:21:20					
Uptime child:	5+17:29:28					
NAME	AVERAGE	AVG_10	AVG_100	AVG_1000	CURRENT	
↪CHANGE						
MAIN.uptime					494968	
↪1.00	1.00	1.00	1.00	1.00		
MAIN.sess_conn					1545	
↪0.00	.	0.00	0.00	0.00		
MAIN.client_req					1569	
↪0.00	.	0.00	0.00	0.00		
MAIN.cache_hit					461	
↪0.00	.	0.00	0.00	0.00		
MAIN.cache_hitpass					16	
↪0.00	.	0.00	0.00	0.00		
MAIN.cache_miss					477	
↪0.00	.	0.00	0.00	0.00		
MAIN.backend_conn					1060	
↪0.00	.	0.00	0.00	0.00		
MAIN.fetch_head					18	
↪0.00	.	0.00	0.00	0.00		
MAIN.fetch_length					996	
↪0.00	.	0.00	0.00	0.00		
MAIN.fetch_204					1	
↪0.00	.	0.00	0.00	0.00		

MAIN.fetch_304				46	
↪0.00	.	0.00	0.00	0.00	
MAIN.pools				9	
↪0.00	.	9.00	9.00	9.00	
MAIN.threads				900	
↪0.00	.	900.00	900.00	900.00	
MAIN.threads_created				900	
↪0.00	.	0.00	0.00	0.00	
...					

Virtual hosting proxy rule

Varnish 4.x example

Varnish 4.x has been released, almost three years after the release of Varnish 3.0 in June 2011. The backend fetching parts of VCL again have changed in Varnish 4.

An example with two separate Plone installations (Zope standalone mode) behind Varnish 4.x HTTP 80 port.

Example:

```
# To make sure that people have upgraded their VCL to the current version,
# Varnish now requires the first line of VCL to indicate the VCL version number
vcl 4.0;

#
# This backend never responds... we get hit in the case of bad virtualhost name
#
backend default {
    .host = "127.0.0.1";
    .port = "55555";
}

#
# Plone Zope clients
#
backend sitel {
    .host = "127.0.0.1";
    .port = "9944";
}

backend site2 {
    .host = "127.0.0.1";
    .port = "9966";
}

#
# Guess which site / virtualhost we are diving into.
# Apache, Nginx or Plone directly
#
sub choose_backend {

    if (req.http.host ~ "^(*.*)?site2\.fi(:[0-9]+)?$") {
        set req.backend_hint = site2;

        # Zope VirtualHostMonster
```

```

        set req.url = "/VirtualHostBase/http/" + req.http.host + ":80/Plone/
↪VirtualHostRoot" + req.url;

    }

    if (req.http.host ~ "^(*.*)?site1\.fi(:[0-9]+)?$") {
        set req.backend_hint = site1;

        # Zope VirtualHostMonster
        set req.url = "/VirtualHostBase/http/" + req.http.host + ":80/Plone/
↪VirtualHostRoot" + req.url;
    }
}

# For now, we'll only allow purges coming from localhost
acl purge {
    "127.0.0.1";
    "localhost";
}

sub vcl_recv {

    #
    # Do Plone cookie sanitization, so cookies do not destroy cacheable anonymous_
↪pages
    #
    if (req.http.Cookie) {
        set req.http.Cookie = ";" + req.http.Cookie;
        set req.http.Cookie = regsuball(req.http.Cookie, "; +", ";");
        set req.http.Cookie = regsuball(req.http.Cookie, "(statusmessages|__ac|_
↪ZopeId|__cp)=", "; \1=");
        set req.http.Cookie = regsuball(req.http.Cookie, "[^ ][^;]*", "");
        set req.http.Cookie = regsuball(req.http.Cookie, "^[; ]+|[; ]+$", "");

        if (req.http.Cookie == "") {
            unset req.http.Cookie;
        }
    }

    call choose_backend;

    if (req.method != "GET" &&
        req.method != "HEAD" &&
        req.method != "PUT" &&
        req.method != "POST" &&
        req.method != "TRACE" &&
        req.method != "OPTIONS" &&
        req.method != "DELETE") {
        /* Non-RFC2616 or CONNECT which is weird. */
        return (pipe);
    }
    if (req.method != "GET" && req.method != "HEAD") {
        /* We only deal with GET and HEAD by default */
        return (pass);
    }
    if (req.http.Authorization || req.http.Cookie) {
        /* Not cacheable by default */

```

```

        return (pass);
    }
    return (hash);
}

sub vcl_hash {
    hash_data(req.url);
    if (req.http.host) {
        hash_data(req.http.host);
    } else {
        hash_data(server.ip);
    }
    return (lookup);
}

# error() is now synth()
sub vcl_synth {
    if (resp.status == 720) {
        # We use this special error status 720 to force redirects with 301
        ↪(permanent) redirects
        # To use this, call the following from anywhere in vcl_recv: error 720 "http://
        ↪/host/new.html"
        set resp.status = 301;
        set resp.http.Location = resp.reason;
        return (deliver);
    } elseif (resp.status == 721) {
        # And we use error status 721 to force redirects with a 302 (temporary)
        ↪redirect
        # To use this, call the following from anywhere in vcl_recv: error 720 "http://
        ↪/host/new.html"
        set resp.status = 302;
        set resp.http.Location = resp.reason;
        return (deliver);
    }

    return (deliver);
}

sub vcl_synth {
    set resp.http.Content-Type = "text/html; charset=utf-8";
    set resp.http.Retry-After = "5";

    synthetic( {"
        <?xml version="1.0" encoding="utf-8"?>
        <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.
        ↪org/TR/xhtml1/DTD/xhtml1-strict.dtd">
        <html>
        <head>
        <title>"} + resp.status + " " + resp.reason + {"</title>
        </head>
        <body>
        <h1>Error "} + resp.status + " " + resp.reason + {"</h1>
        <p>"} + resp.reason + {"</p>
        <h3>Guru Meditation:</h3>
        <p>XID: "} + req.xid + {"</p>
        <hr>
        <p>Varnish cache server</p>
        </body>
    
```

```
        </html>

    " } );

    return (deliver);
}
```

Load balancing

Load balancing increases performance and resilience. Varnish `vmod_directors` module enables load balancing using a concept called “directors”.

A director is a group of several backend servers. A backend server is a server providing the content Varnish will accelerate.

Varnish supports the following directors:

round-robin Picks backends in a round-robin fashion.

fallback Try each of the added backends in turn, and return the first one that is healthy.

hash Chooses the backend server by computing the hash of a string.

random Distributes load over the backends using a weighted random probability distribution.

The following example shows how to configure round-robin load balancing of 2 Plone instances.

```
import directors;

backend instance1 {
    .host = "localhost";
    .port = "8081";
}

backend instance2 {
    .host = "localhost";
    .port = "8082";
}

sub vcl_init {
    new plone = directors.round_robin();
    plone.add_backend(instance1);
    plone.add_backend(instance2);
}

sub vcl_recv {
    set req.backend_hint = plone.backend();
}
```

For more information, see:

- <https://www.varnish-cache.org/docs/trunk/users-guide/vcl-backends.html>
- https://www.varnish-cache.org/docs/trunk/reference/vmod_directors.generated.html

Varnishd port and IP address to listen

You give IP address(s) and ports for Varnish to listen to on the `varnishd` command line using `-a` switch. Edit `/etc/default/varnish`:


```
DAEMON_OPTS="-a 192.168.1.1:80 \
             -T localhost:6082 \
             -f /etc/varnish/default.vcl \
             -s file,/var/lib/varnish/$INSTANCE/varnish_storage.bin,1G"
```

Cached and editor sub domains

You can provide an uncached version of the site for editors:

- <http://serverfault.com/questions/297541/varnish-cached-and-non-cached-subdomains/297547#297547>

Sanitizing cookies

Any cookie set on the server side (session cookie) or on the client-side (e.g. Google Analytics JavaScript cookies) is poison for caching the anonymous visitor content.

HTTP caching needs to deal with both HTTP request and response cookie handling

- HTTP request *Cookie* header. The browser sending HTTP request with *Cookie* header confuses Varnish cache look-up. This header can be set by JavaScript also, not just by the server. *Cookie* can be preprocessed in varnish's `vcl_recv` step.
- HTTP response *Set-Cookie* header. This sets a server-side cookie. If your server is setting cookies Varnish does not cache these responses by default. However, this might be desirable behavior if e.g. multi-lingual content is served from one URL with language cookies. *Set-Cookie* can be post-processed in varnish's `vcl_fetch` step.

Example of removing all Plone-related cookies, besides ones dealing with the logged in users (content authors):

```
sub vcl_recv {

    if (req.http.Cookie) {
        # (logged in user, status message - NO session storage or language cookie)
        set req.http.Cookie = ";" + req.http.Cookie;
        set req.http.Cookie = regsuball(req.http.Cookie, "; +", ";");
        set req.http.Cookie = regsuball(req.http.Cookie, "(statusmessages|__ac|_
→ZopeId|__cp)=", "; \1=");
        set req.http.Cookie = regsuball(req.http.Cookie, "[^ ][^;]*", "");
        set req.http.Cookie = regsuball(req.http.Cookie, "^[; ]+|[; ]+$", "");

        if (req.http.Cookie == "") {
            unset req.http.Cookie;
        }
    }
    ...

sub vcl_backend_response {

    # Here we could unset cookies explicitly,
    # but we assume plone.app.caching extension does it jobs
    # and no extra cookies fall through for HTTP responses we'd like to cache
    # (like images)

    if (beresp.ttl <= 0s
        || beresp.http.Set-Cookie
        || beresp.http.Surrogate-control ~ "no-store"
```

```
    || (!beresp.http.Surrogate-Control && beresp.http.Cache-Control ~ "no-
↪cache|no-store|private")
    || beresp.http.Vary == "*") {
        /* * Mark as "Hit-For-Pass" for the next 2 minutes */
        set beresp.ttl = 120s;
        set beresp.uncacheable = true;
    }

    set beresp.grace = 120s;
    return (deliver);
}
```

An example how to purge Google cookies only and allow other cookies by default:

```
sub vcl_recv {
    # Remove Google Analytics cookies - will prevent caching of anon content
    # when using GA JavaScript. Also you will lose the information of
    # time spend on the site etc..
    if (req.http.cookie) {
        set req.http.Cookie = regsuball(req.http.Cookie, "__utm=[^;]+(; )?", "");
        if (req.http.cookie ~ "^*$") {
            unset req.http.cookie;
        }
    }
    ....
}
```

Debugging cookie issues

Use the following snippet to set a HTTP response debug header to see what the backend server sees as cookie after `vcl_recv` clean-up regexes:

```
sub vcl_backend_response {

    /* Use to see what cookies go through our filtering code to the server */
    set beresp.http.X-Varnish-Cookie-Debug = "Cleaned request cookie: " + req.http.
↪Cookie;

    if (beresp.ttl <= 0s ||
        beresp.http.Set-Cookie ||
        beresp.http.Vary == "*") {
        /*
         * Mark as "Hit-For-Pass" for the next 2 minutes
         */
        # hit_for_pass objects are created using beresp.uncacheable
        set beresp.uncacheable = true;
        set beresp.ttl = 120s;
        return (deliver);
    }
}
```

And then test with `wget`:

```
cd /tmp # wget wants to save files...
wget -S http://www.site.fi
--2011-11-16 11:28:37--  http://www.site.fi/
Resolving www.site.fi (www.site.fi)... xx.20.128.xx
```

```

Connecting to www.site.fi (www.site.fi)|xx.20.128.xx|:80... connected.
HTTP request sent, awaiting response...
HTTP/1.1 200 OK
Server: Zope/(2.12.17, python 2.6.6, linux2) ZServer/1.1
X-Cache-Operation: plone.app.caching.noCaching
Content-Language: fi
Expires: Sun, 18 Nov 2001 09:28:37 GMT
Cache-Control: max-age=0, must-revalidate, private
X-Cache-Rule: plone.content.folderView
Content-Type: text/html;charset=utf-8
Set-Cookie: I18N_LANGUAGE="fi"; Path=/
Content-Length: 23836
X-Varnish-Cookie-Debug:Cleaned request cookie: __
↪gads=ID=1477fbe04d35a542:T=1405963607:S=ALNI_MYJat5RSzKvD5xve78jLJsxl6-b_Q; __ac=
↪"NjE2NDZkNjk2ZTo2NDMxMzQyNDcwMzQ3MjMwNmMzMjc2MzM3Mg%253D%253D"
Date: Wed, 16 Nov 2011 09:28:37 GMT
X-Varnish: 1562749485
Age: 0
Via: 1.1 varnish-v4

```

More info

- <https://www.varnish-software.com/blog/adding-headers-gain-insight-vcl>

Plone Language cookie (I18N_LANGUAGE)

This cookie could be removed in `vcl_fetch` response post-processing (how?). However, a better solution is to disable this cookie in the backend itself: in this case in Plone's `portal_languages` tool. Disable it by *Use cookie for manual override* setting in `portal_languages`.

More info

- [Plone cookies documentation](#)
- <https://www.varnish-cache.org/docs/4.0/users-guide/increasing-your-hitrate.html#cookies>

Do not cache error pages

You can make sure that Varnish does not accidentally cache error pages. E.g. it would cache front page when the site is down:

```

sub vcl_backend_response {
    if (beresp.status >= 500 && beresp.status < 600) {
        unset beresp.http.Cache-Control;
        set beresp.http.Cache-Control = "no-cache, max-age=0, must-revalidate";
        set beresp.ttl = 0s;
        set beresp.http.Pragma = "no-cache";
        set beresp.uncacheable = true;
        return(deliver);
    }
    ...
}

```

Custom and full cache purges

Below is an example how to create an action to purge the whole Varnish cache.

First you need to allow HTTP PURGE request in `default.vcl` from `localhost`. We'll create a special PURGE command which takes URLs to be purged out of the cache in a special header:

```
acl purge {
    "localhost";
    # XXX: Add your local computer public IP here if you
    # want to test the code against the production server
    # from the development instance
}
...

sub vcl_recv {
    ...
    # Allow PURGE requests clearing everything
    if (req.method == "PURGE") {
        if (!client.ip ~ purge) {
            return(synth(405, "Not allowed.));
        }
        return (purge);
    }
}
```

Then let's create a Plone view which will make a request from Plone to Varnish (upstream `localhost:80`) and issue the PURGE command. We do this using the [Requests](#) Python library.

Example view code:

```
import requests

from Products.Five import BrowserView

from requests.models import Request

class Purge(BrowserView):
    """
    Purge upstream cache from all entries.

    This is ideal to hook up for admins e.g. through portal_actions menu.

    You can access it as admin::

        http://site.com/@@purge

    """

    def __call__(self):
        """
        Call the parent cache using Requests Python library and issue PURGE command
        ↪ for all URLs.

        Pipe through the response as is.
        """

        # This is the root URL which will be purged
        # - you might want to have different value here if
```

```

# your site has different URLs for manage and themed versions
site_url = self.context.portal_url() + "/"

headers = {
    # Match all pages
    "X-Purge-Regex" : ".*"
}

resp = requests.request("PURGE", site_url + "*", headers=headers)

self.request.response["Content-type"] = "text/plain"
text = []

text.append("HTTP " + str(resp.status_code))

# Dump response headers as is to the Plone user,
# so he/she can diagnose the problem
for key, value in resp.headers.items():
    text.append(str(key) + ": " + str(value))

# Add payload message from the server (if any)

if hasattr(resp, "body"):
    text.append(str(resp.body))

```

Registering the view in ZCML:

```

<browser:view
    for="Products.CMFPlone.interfaces.IPloneSiteRoot"
    name="purge"
    class=".views.Purge"
    permission="cmf.ManagePortal"
/>

```

More info

- <https://www.varnish-cache.org/docs/4.0/users-guide/purging.html>

Frontend Webserver

Description

How to configure the most popular Frontend servers.

This guide particularly focuses on [Unix-like](#) environments, though the stack discussion may be useful to everyone.

Apache

Description

Tips and guides for hosting Plone with Apache web server.

Introduction

Here are useful information and snippets when hosting Plone behind Apache.

Installing Apache front-end for Plone

Apache runs on port 80. Plone runs on port 8080. Apache accepts all HTTP traffic to your internet domain.

Here are quick instructions for Ubuntu / Debian.

Install required software:

```
sudo apt-get install apache2
sudo a2enmod rewrite
sudo a2enmod proxy
sudo a2enmod proxy_http
sudo a2enmod headers
sudo /etc/init.d/apache2 restart
```

Add virtual host config file `/etc/apache2/sites-enabled/yoursite.conf`. Assuming *Plone* is your site id in the Management Interface (capital lettering do matter) and your domain name is `yoursite.com` (note with or without `www` matters, see below):

```
UseCanonicalName On

NameVirtualHost *
<VirtualHost *>
    ServerAlias yoursite.com
    ServerSignature On

    Header set X-Frame-Options "SAMEORIGIN"
    Header set Strict-Transport-Security "max-age=15768000; includeSubDomains"
    Header set X-XSS-Protection "1; mode=block"
    Header set X-Content-Type-Options "nosniff"
    Header set Content-Security-Policy-Report-Only "default-src 'self'; img-src *;
↪style-src 'unsafe-inline'; script-src 'unsafe-inline' 'unsafe-eval'"

    ProxyVia On

    # prevent your web server from being used as global HTTP proxy
    <LocationMatch "^[/]">
        Deny from all
    </LocationMatch>

    <Proxy *>
        Order deny,allow
        Allow from all
    </Proxy>

    RewriteEngine on
    RewriteRule ^/(.*) http://localhost:8080/VirtualHostBase/http/${HTTP_HOST}:80/
↪Plone/VirtualHostRoot/$1 [P,L]
</VirtualHost>

<VirtualHost *>
    ServerAlias *
    ServerRoot /var/www
```

```
ServerSignature On
</VirtualHost>
```

Eventually you have one virtual host configuration file per one domain on your server.

Restart apache:

```
sudo apache2ctl configtest
sudo apache2ctl restart
```

Check that Plone responds:

```
http://yoursite.com:8080/Plone
```

Check that Apache responds:

```
http://yoursite.com
```

If everything is good then your Plone site properly configured using Apache front-end.

Content Security Policy (CSP) prevents a wide range of attacks, including cross-site scripting and other cross-site injections, but the CSP header setting may require careful tuning. To enable it, replace the Content-Security-Policy-Report-Only by Content-Security-Policy. The example above works with Plone 5.x (including TinyMCE) but it very wide. You may need to adjust it if you want to make CSP more restrictive or use additional Plone Products. For more information, see

- <http://www.w3.org/TR/CSP/>

For an SSL configuration, just modify the rewrite rule from

```
RewriteRule ^/(.*) http://localhost:8080/VirtualHostBase/http/${HTTP_HOST}:80/Plone/
↳VirtualHostRoot/$1 [P,L]
```

to

```
RewriteRule ^/(.*) http://localhost:8080/VirtualHostBase/https/${HTTP_HOST}:443/Plone/
↳VirtualHostRoot/$1 [P,L]
```

inside an SSL-enabled Apache virtual host definition.

Apache and Plone guide (old)

Procedure to restart Apache in production environment

You might share the same Apache web server across several production sites. You don't want to hinder the performance of the other sites when doing Apache configuration changes to one site.

The correct procedure to restart Apache is (on Ubuntu/Debian Linux)

```
# Check that config files are working after editing them
apache2ctl configtest

# Let Apache finish serving all the on-going requests before
# restarting worker processes
apache2ctl graceful
```

www-redirects

If you wish to force people to use your site with or without www prefix you can use the rules below. Note that setting this kind of rule is very useful from the search engine optimization point of view also.

Example in <VirtualHost> section to redirect `www.site.com` -> `site.com`

```
<VirtualHost 127.0.0.1:80>

    ServerName site.com
    ServerAlias www.site.com

    <IfModule mod_rewrite.c>
        RewriteEngine On
        RewriteCond %{HTTP_HOST} ^www\.site\.com [NC]
        RewriteRule (.*) http://site.com$1 [L,R=302]

    </IfModule>
```

Example in <VirtualHost> section to redirect `site.com` -> `www.site.com`

```
<VirtualHost 127.0.0.1:80>

    ServerName site.com
    ServerAlias www.site.com

    <IfModule mod_rewrite.c>
        RewriteEngine On
        RewriteCond %{HTTP_HOST} ^site\.com [NC]
        RewriteRule (.*) http://www.site.com$1 [L,R=302]

    </IfModule>
```

Redirecting all the pages to the root of a new site:

```
RewriteEngine On RewriteRule (.*) http://www.newsite.com [L,R=302]
```

Migration redirects

To redirect traffic from all pages permanently (301) to the landing page of a new site:

```
RewriteEngine On
RewriteRule (.*) http://docs.plone.org/ [L,R=301]
```

Proxying other site under Plone URI space

The following rule can be used to put a static web site to sit in the same URI space with Plone. Put these rules **before** VirtualHost ProxyPass.

Examples:

```
ProxyPass /othersite/ http://www.some.other.domain.com/othersite/
ProxyPassReverse /othersite/ http://www.some.other.domain.com/othersite/
```


Reverse proxy host

By default, host name is correctly delivered from Apache to Plone. Otherwise you might see all your HTTP requests coming from localhost, Apache.

You need

```
ProxyPreserveHost On
```

For more information, see

- <http://macadames.wordpress.com/2009/05/23/some-deliverance-tips/>

Redirecting certain URIs to old site

This is useful if you migrate to a Plone from some legacy technology and you still need to have some part of the URI space to point to the old server.

- Create alternative domain name for the existing old site (e.g. www2)
- Modify Apache configuration so that URLs still being used are redirected to the old server with alternative name, Put in this rewrite

```
<location /media>
    RedirectMatch /media/(.*)$ http://www2.site.fi/media/$1
</location>
```

Virtual hosting Apache configuration generator

- <http://betabug.ch/zope/witch>

Caching images

There are much better caching solutions for Plone than Apache's mod_cache, see the *Guide to caching*.

One important thing to know about mod_cache is that by default it caches Set-Cookie headers. Most likely, this is not what you want when using it with Plone, so you should use the CacheIgnoreHeaders directive to strip Set-Cookie headers from cached objects.

Have a close look at the official [Apache documentation](#)) and also read the comments at the bottom, they are very informative - even more so in the [2.2 version](#).

If you cannot avoid using mod_cache, you can configure disk based Apache caching as follows:

First you need to enable the relevant Apache modules:

```
* mod_cache, mod_diskcache
```

On Debian this is

```
sudo a2enmod
```

Then you can add to your virtual host configuration:

```
# Disk cache configuration
CacheEnable disk /
CacheRoot "/var/cache/yourorganization-production"
CacheLastModifiedFactor 0.1
#CacheDefaultExpire 1
#CacheMaxExpire 7200
CacheDirLength 2
# the next line is important, see above
CacheIgnoreHeaders Set-Cookie
```

Then go to *Cache Configuration* (Plone 4.1+) and configure [the caching options](#).

Testing cache headers

Use UNIX *wget* command. *-S* flag will display request headers.

Remember to do different request for HTML, CSS, JS and image payloads - the cache rules might not be the same.

HTTP example:

```
cd /tmp

wget --cache=off -S http://production.yourorganizationinternational.org/
↳yourorganizationlogotemplate.gif
```

```
HTTP request sent, awaiting response...
HTTP/1.1 200 OK
Date: Tue, 09 Mar 2010 12:33:26 GMT
Server: Apache/2.2.8 (Ubuntu) DAV/2 SVN/1.4.6 mod_python/3.3.1 Python/2.5.2 PHP/5.2.
↳4-2ubuntu5.4 with Suhosin-Patch mod_ssl/2.2.8 OpenSSL/0.9.8g
Last-Modified: Wed, 25 Nov 2009 06:51:41 GMT
Content-Length: 4837
Via: 1.0 production.yourorganizationinternational.org
Cache-Control: max-age=3600, public
Expires: Tue, 09 Mar 2010 13:02:29 GMT
Age: 1857
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: image/gif
Length: 4837 (4.7K) [image/gif]
Saving to: `yourorganizationlogotemplate.gif.14'
```

HTTPS example:

```
cd /tmp

wget --cache=off --no-check-certificate -S https://production.
↳yourorganizationinternational.org/
```

Flushing cache

Manually cleaning Apache disk cache:

```
sudo -i
cd /var/cache/yoursite
rm -rf *
```

Custom 500 internal error page

To make you look more pro when you update the server or Plone goes down

- <https://httpd.apache.org/docs/2.2/custom-error.html>

Load balanced Apache virtual host configuration

This complex config example includes

- HTTPS and SSL certificate set-up
- Load balancing using ZEO front-ends and Apache load balancer module
- Apache disk cache. This should provide static resource caching w/HTTPS support if you are using plone.app.caching.
- <https://httpd.apache.org/docs/2.2/caching.html>

See

- <http://stackoverflow.com/questions/5650716/are-sticky-sessions-needed-when-load-balancing-plone-3-3-5>

More information about how to set a sticky session cookie if you need to support Zope sessions in your code

- http://opensourcehacker.com/2011/04/15/sticky-session-load-balancing-with-apache-and-mod_balancer-on-ubuntu-linux/

Example:

```
<VirtualHost 123.123.123.123:443>

    ServerName  production.yourorganization.org
    ServerAdmin rocks@mfabrik.com

    SSLEngine On
    SSLCertificateFile /etc/apache2/ssl-keys/yourorganization.org.cer
    SSLCertificateKeyFile /etc/apache2/ssl-keys/yourorganization.org.key
    SSLCertificateChainFile /etc/apache2/ssl-keys/InstantValidationCertChain.crt

    LogFormat      combined
    TransferLog     /var/log/apache2/production.yourorganization.org.log

    <IfModule mod_proxy.c>
        ProxyVia On

        # prevent the webserver from being used as proxy
        <LocationMatch "^[/]">
            Deny from all
        </LocationMatch>
    </IfModule>

    # Balance load between 4 ZEO front-ends
    <Proxy balancer://lbyourorganization>
        BalancerMember http://127.0.0.1:13001/
        BalancerMember http://127.0.0.1:13002/
        BalancerMember http://127.0.0.1:13003/
        BalancerMember http://127.0.0.1:13004/
        # Use Pending Request Counting Algorithm (s. http://httpd.apache.org/docs/current/
        ↪ mod/mod_lbmethod_bybusyness.html).
```

```
# This will reduce latencies that occur as a result of long running requests,
↳ temporarily blocking a ZEO client.
# You will need to install the separate mod_lbmethod_bybusyness module in Apache,
↳ 2.4.
ProxySet lbmethod=bybusyness
</Proxy>

# Note: You might want to disable this URL of being public
# as it can be used to access Apache live settings
<Location /balancer-manager>
    SetHandler balancer-manager
    Order Deny,Allow
    # Your trusted IP addresses
    Allow from 123.123.123.123
</Location>

ProxyPass /balancer-manager !
ProxyPass          / balancer://lbyourorganization/http://localhost/
↳ VirtualHostBase/https/production.yourorganization.org:443/yourorganization_plone_
↳ site/VirtualHostRoot/
ProxyPassReverse    / balancer://lbyourorganization/http://localhost/
↳ VirtualHostBase/https/production.yourorganization.org:443/yourorganization_plone_
↳ site/VirtualHostRoot/

# Disk cache configuration, if you really must use Apache for caching
CacheEnable disk /
# Must point to www-data writable directly which depends on OS
CacheRoot "/var/cache/yourorganization-production"
CacheLastModifiedFactor 0.1
CacheIgnoreHeaders Set-Cookie

# Debug header flags all requests coming from this server
Header append X-YourOrganization-Production yes

</VirtualHost>
```

Enabling gzip compression

Enabling gzip compression in Apache will make your web sites respond much more quickly for your web site users and will reduce the amount of bandwidth used by your web sites.

Instructions for enabling gzip in Apache:

- <https://varvy.com/pagespeed/enable-compression.html>
- http://httpd.apache.org/docs/2.2/mod/mod_deflate.html

Nginx

Description

Using the nginx web server to host Plone sites

Introduction

Nginx is an modern alternative server to Apache.

- It acts as a proxy server and load balancer in front of Zope.
- It handles rewrite rules.
- It handles HTTPS.

Minimal Nginx front end configuration for Plone on Ubuntu/Debian Linux

This is a minimal configuration to run nginx on Ubuntu/Debian in front of a Plone site. These instructions are *not* for configurations where one uses the buildout configuration tool to build a static Nginx server.

- Plone will by default be served on port 8080.
- We use [VirtualHostMonster](#) to pass the original protocol and hostname to Plone. VirtualHostMonster provides a way to rewrite the request path.
- We also need to rewrite the request path, because you want to site be served from port 80 root (/), but Plone sites are nested in the Zope application server as paths */site1*, */site2* etc.
- You don't need to configure VirtualHostMonster in Plone/Zope in any way, because all the installers will automatically install one for you. Nginx configuration is all you need to touch.
- The URL passed to VirtualHostMonster is the URL Plone uses to construct links in the template (`portal_url` in the code, also used by `content.absolute_url()` method). If your site loads without CSS styles usually it is a sign that VirtualHostMonster URL is incorrectly written – Plone uses the URL to link stylesheets also.
- Plone itself contains a mini web server (Medusa) which serves the requests from port 8080 – Nginx acts simple as a HTTP proxy between Medusa and outgoing port 80 traffic. Nginx does not spawn Plone process or anything like that, but Plone processes are externally controlled, usually by buildout-created `bin/instance` and `bin/plonectl` commands, or by a supervisor instance.

Create file `/etc/nginx/sites-available/yoursite.conf` with contents:

```
# This adds security headers
add_header X-Frame-Options "SAMEORIGIN";
add_header Strict-Transport-Security "max-age=15768000; includeSubDomains";
add_header X-XSS-Protection "1; mode=block";
add_header X-Content-Type-Options "nosniff";
#add_header Content-Security-Policy "default-src 'self'; img-src *; style-src 'self'
→'unsafe-inline'; script-src 'self' 'unsafe-inline' 'unsafe-eval'";
add_header Content-Security-Policy-Report-Only "default-src 'self'; img-src *; style-
→src 'self' 'unsafe-inline'; script-src 'self' 'unsafe-inline' 'unsafe-eval'";

# This specifies which IP and port Plone is running on.
# The default is 127.0.0.1:8080
upstream plone {
    server 127.0.0.1:8090;
}

# Redirect all www-less traffic to the www.site.com domain
# (you could also do the opposite www -> non-www domain)
server {
    listen 80;
    server_name yoursite.com;
    rewrite ^/(.*) http://www.yoursite.com/$1 permanent;
```

```
}

server {

    listen 80;
    server_name www.yoursite.com;
    access_log /var/log/nginx/yoursite.com.access.log;
    error_log /var/log/nginx/yoursite.com.error.log;

    # Note that domain name spelling in VirtualHostBase URL matters
    # -> this is what Plone sees as the "real" HTTP request URL.
    # "Plone" in the URL is your site id (case sensitive)
    location / {
        proxy_pass http://plone/VirtualHostBase/http/yoursite.com:80/Plone/
↪VirtualHostRoot/;
    }
}
```

Then enable the site by creating a symbolic link:

```
sudo -i
cd /etc/nginx/sites-enabled
ln -s ../sites-available/yoursite.conf .
```

See that your nginx configuration is valid:

```
/etc/init.d/nginx configtest

ok
configuration file /etc/nginx/nginx.conf test is successful
nginx.
```

Alternatively your system might not provide `configtest` command and then you can test config with:

```
/usr/sbin/nginx
```

If the config was OK then restart:

```
/etc/init.d/nginx restart
```

More info:

- http://wiki.mediatemple.net/w/%28ve%29:Configure_virtual_hosts_with_Nginx_on_Ubuntu
- <http://www.starzel.de/blog/securing-plone-sites-with-https-and-nginx>

Content Security Policy (CSP) prevents a wide range of attacks, including cross-site scripting and other cross-site injections, but the CSP header setting may require careful tuning. To enable it, replace the Content-Security-Policy-Report-Only by Content-Security-Policy. The example above works with Plone 4.x and up (including TinyMCE) but it very wide. You may need to adjust it if you want to make CSP more restrictive or use additional Plone Products. For more information, see

- <http://www.w3.org/TR/CSP/>

Buildout and recipe

If, and only if, you cannot use a platform install of nginx you may use the recipe and buildout example below to get started.

- <http://www.martinaspeli.net/articles/an-uber-buildout-for-a-production-plone-server>
- <https://pypi.python.org/pypi/gocept.nginx>

A buildout will download, install and configure nginx from scratch. The buildout file contains an nginx configuration which can use template variables from `buildout.cfg` itself.

When you change the configuration of nginx in buildout you probably don't want to rerun the whole buildout, but only the nginx part of it:

```
bin/buildout -c production.cfg install balancer
```

Config test

Assuming you have a buildout nginx section called `balancer`:

```
bin/balancer configtest

Testing nginx configuration
the configuration file /srv/plone/isleofback/parts/balancer/balancer.conf syntax is ok
configuration file /srv/plone/isleofback/parts/balancer/balancer.conf test is_
↪successful
```

Deployment configuration

`gocept.nginx` supports a special deployment configuration where you manually configure all directories. One important reason why you might wish to do this, is to change the location of the `pid` file. Normally this file would be created in `parts`, which is deleted and recreated when you re-run buildout. This interferes with reliably restarting nginx, since the `pid` file may have been deleted since startup. In this case, you need to manually kill nginx to get things back on track.

Example deployment configuration in `production.cfg`:

```
# Define folder and file locations for nginx called "balancer"
# If deployment= is set on gocept.nginx recipe it uses
# data provider here
[nginx]
run-directory = ${buildout:directory}/var/nginx
etc-directory = ${buildout:directory}/var/nginx
log-directory = ${buildout:directory}/var/logs
rc-directory = ${buildout:directory}/bin
logrotate-directory =
user =

[balancer]
recipe = gocept.nginx
nginx = nginx-build
deployment = nginx
configuration =
    #user ${users:balancer};
    error_log ${buildout:directory}/var/log/balancer-error.log;
    worker_processes 1;
```

Install this part:

```
bin/buildout -c production.cfg install balancer
```

Then you can use the following cycle to update the configuration:

```
bin/balancer-nginx-balancer start
# Update config in buildout
nano production.cfg
# This is non-destructive, because now our PID file is in var/nginx
bin/buildout -c production.cfg install balancer
# Looks like reload is not enough
bin/nginx-balancer stop ; bin/nginx-balancer start
```

Manually killing nginx

You have lost PID file, or the recorded PID does not match the real PID any longer. Use buildout's starter script as a search key:

```
(hardy_i386)isleofback@isleofback:~$ bin/balancer reload
Reloading nginx
cat: /srv/plone/isleofback/parts/balancer/balancer.pid: No such file or directory

(hardy_i386)isleofback@isleofback:~$ ps -Af|grep -i balancer
1001      14012      1  0 15:26 ?          00:00:00 nginx: master process /srv/plone/
↪isleofback/parts/nginx-build/sbin/nginx -c /srv/plone/isleofback/parts/balancer/
↪balancer.conf
1001      16488 16458  0 16:34 pts/2      00:00:00 grep -i balancer
(hardy_i386)isleofback@isleofback:~$ kill 14012

# balancer is no longer running
(hardy_i386)isleofback@isleofback:~$ ps -Af|grep -i balancer
1001      16496 16458  0 16:34 pts/2      00:00:00 grep -i balancer

(hardy_i386)isleofback@isleofback:~$ bin/balancer start
Starting nginx

# Now it is running again
(hardy_i386)isleofback@isleofback:~$ ps -Af|grep -i balancer
1001      16501      1  0 16:34 ?          00:00:00 nginx: master process /srv/plone/
↪isleofback/parts/nginx-build/sbin/nginx -c /srv/plone/isleofback/parts/balancer/
↪balancer.conf
1001      16504 16458  0 16:34 pts/2      00:00:00 grep -i balancer
```

Debugging nginx

Set nginx logging to debug mode:

```
error_log ${buildout:directory}/var/log/balancer-error.log debug;
```

www-redirect

Below is an example how to do a basic *yourdomain.com* -> *www.yourdomain.com* redirect.

Put the following in your `gocept.nginx` configuration:


```
http {
    ....
    server {
        listen ${hosts:balancer}:${ports:balancer};
        server_name ${hosts:main-alias};
        access_log off;
        rewrite ^(.*)$ $scheme://${hosts:main}$1 redirect;
    }
}
```

Hosts are configured in a separate buildout section:

```
[hosts]
# Hostnames for servers
main = www.yoursite.com
main-alias = yoursite.com
```

More info

- <https://stackoverflow.com/questions/7947030/nginx-no-www-to-www-and-www-to-no-www>

Permanent redirect

Below is an example redirect rule:

```
# Redirect old Google front page links.
# Redirect event to new Plone based systems.

location /tapahtumat.php {
    rewrite ^ http://${hosts:main}/tapahtumat permanent;
}
```

Note: Nginx location match evaluation rules are not always top-down. You can add more specific matches after location /.

Cleaning up query string

By default, nginx includes all trailing HTTP GET query parameters in the redirect. You can disable this behavior by adding a trailing ?:

```
location /tapahtumat.php {
    rewrite ^ http://${hosts:main}/no_ugly_query_string? permanent;
}
```

Matching incoming query string

The `location` directive does not support query strings. Use the `if` directive from the HTTP rewrite module.

Example:

```
location /index.php {
    # index.php?id=5
```

```
        if ($args ~ id=5) {
            rewrite ^ http://${hosts:main}/sisalto/lomapalvelut/ruokailu?_
↪permanent;
        }
    }
```

More info

nginx location matching rules

- <http://wiki.nginx.org/NginxHttpCoreModule#location>

nginx redirect module docs

- <http://wiki.nginx.org/NginxHttpRewriteModule>

More info on nginx redirects

- <http://scott.yang.id.au/2007/04/do-you-need-permalink-redirect/>

Make nginx aware where the request came from

If you set up nginx to run in front of Zope, and set up a virtual host with it like this:

```
server {
    server_name demo.webandmobile.mfabrik.com;
    location / {
        rewrite ^/(.*)$ /VirtualHostBase/http/demo.webandmobile.mfabrik.
↪com:80/Plone/VirtualHostRoot/$1 break;
        proxy_pass http://127.0.0.1:8080/;
    }
}
```

Zope will always get the request from 127.0.0.1:8080 and not from the actual host, due to the redirection. To solve this problem correct your configuration to be like this:

```
server {
    server_name demo.webandmobile.mfabrik.com;
    location / {
        rewrite ^/(.*)$ /VirtualHostBase/http/demo.webandmobile.mfabrik.
↪com:80/Plone/VirtualHostRoot/$1 break;
        proxy_pass http://127.0.0.1:8080/;
        proxy_set_header    Host                $host;
        proxy_set_header    X-Real-IP           $remote_addr;
        proxy_set_header    X-Forwarded-For    $proxy_add_x_forwarded_for;
    }
}
```

SSI: server-side include

In order to include external content in a page (XDV), we must set up nginx to make these includes for us. For including external content we will use the SSI (server-side include) method, which means that on each request nginx will get the needed external content, put it in place and only then return the response. Here is a configuration that sets up the filtering and turns on SSI for a specific location:

```

server {
    listen 80;
    server_name localhost;

    # Decide if we need to filter
    if ($args ~ "^(.*)filter_xpath=(.*)$") {
        set $newargs $1;
        set $filter_xpath $2;
        # rewrite args to avoid looping
        rewrite    ^(.*)$    /_include$1?$newargs?;
    }

    location @include500 { return 500; }
    location @include404 { return 404; }

    location ^~ /_include {
        # Restrict to subrequests
        internal;
        error_page 404 = @include404;

        # Cache in Varnish for 1h
        expires 1h;

        # Proxy
        rewrite    ^/_include(.*)$    $1    break;
        proxy_pass http://127.0.0.1:80;

        # Our safety belt.
        proxy_set_header X-Loop 1$http_X_Loop; # unary count
        proxy_set_header Accept-Encoding "";
        error_page 500 = @include500;
        if ($http_X_Loop ~ "11111") {
            return 500;
        }

        # Filter by xpath
        xslt_stylesheet /home/ubuntu/plone/eggs/xdv-0.4b2-py2.6.egg/xdv/filter.xsl
        xpath=$filter_xpath
        ;
        xslt_html_parser on;
        xslt_types text/html;
    }

    location /forum {
        xslt_stylesheet /home/ubuntu/plone/theme/theme.xsl
        path='$uri'
        ;
        xslt_html_parser on;
        xslt_types text/html;
        # Switch on ssi here to enable external includes.
        ssi on;

        root    /home/ubuntu/phpBB3;
        index    index.php;
        try_files $uri $uri/ /index.php?q=$uri&$args;
    }
}

```

Session affinity

If you intend to use nginx for session balancing between ZEO processes, you need to be aware of session affinity. By default, ZEO processes don't share session data. If you have site functionality which stores user-specific data on the server, let's say an ecommerce site shopping cart, you must always redirect users to the same ZEO client process or they will have 1/number of processes chance to see the original data.

Make sure that your *Zope session cookie* are not cleared by any front-end server (nginx, Varnish).

By using IP addresses

This is the most reliable way. nginx will balance each incoming request to a front end client by the request's source IP address.

This method is reliable as long as nginx can correctly extract IP address from the configuration.

- http://wiki.nginx.org/NginxHttpUpstreamModule#ip_hash

By using cookies

These instructions assume you are installing nginx via buildout.

- [Nginx sticky sessions module](#)

Manually extract nginx-sticky-module under src:

```
cd src
wget https://code.google.com/p/nginx-sticky-module/downloads/list
```

Then add it to the nginx-build part in buildout:

```
[nginx-build]
recipe = zc.recipe.cmmi
url = http://sysoev.ru/nginx/nginx-0.7.65.tar.gz
extra_options = --add-module=${buildout:directory}/src/nginx-sticky-module-1.0-rc2
```

Now test reinstalling nginx in buildout:

```
mv parts/nginx-build/ parts/nginx-build-old # Make sure full rebuild is done
bin/buildout install nginx-build
```

See that it compiles without errors. Here is the line of compiling sticky:

```
gcc -c -O -pipe -O -W -Wall -Wpointer-arith -Wno-unused-parameter \
-Wunused-function -Wunused-variable -Wunused-value -Werror -g \
-I src/core -I src/event -I src/event/modules -I src/os/unix \
-I objs -I src/http -I src/http/modules -I src/mail \
-o objs/addon/nginx-sticky-module-1.0-rc2/nginx_http_sticky_module.o
```

Now add sticky to the load-balancer section of nginx config:

```
[balancer]
recipe = gocept.nginx
nginx = nginx-build
...
http {
```

```

client_max_body_size 64M;
upstream zope {
    sticky;
    server ${hosts:client1}:${ports:client1} max_fails=3 fail_timeout=30s;
    server ${hosts:client2}:${ports:client2} max_fails=3 fail_timeout=30s;
    server ${hosts:client3}:${ports:client3} max_fails=3 fail_timeout=30s;
}

```

Reinstall nginx balancer configs and start-up scripts:

```
bin/buildout install balancer
```

Make sure that the generated configuration is ok:

```
bin/nginx-balancer configtest
```

Restart nginx:

```
bin/nginx-balancer stop ;bin/nginx-balancer start
```

Check that some (non-anonymous) page has the route cookie set:

```

Huiske-iMac:tmp moo$ wget -S http://yoursite.com/sisalto/saariselka-infoa
--2011-03-21 21:31:40-- http://yoursite.com/sisalto/saariselka-infoa
Resolving yoursite.com (yoursite.com)... 12.12.12.12
Connecting to yoursite.com (yoursite.com)|12.12.12.12|:80... connected.
HTTP request sent, awaiting response...
HTTP/1.1 200 OK
Server: nginx/0.7.65
Content-Type: text/html; charset=utf-8
Set-Cookie: route=7136de9c531fcd112f24c3f32c3f52f
Content-Language: fi
Expires: Sat, 1 Jan 2000 00:00:00 GMT
Set-Cookie: I18N_LANGUAGE="fi"; Path=/
Content-Length: 41471
Date: Mon, 21 Mar 2011 19:31:40 GMT
X-Varnish: 1979481774
Age: 0
Via: 1.1 varnish
Connection: keep-alive

```

Now test it by doing session-related activity and see that your shopping cart is not “lost”.

More info

- <http://code.google.com/p/nginx-sticky-module/source/browse/trunk/README>
- <http://nathanvangheem.com/news/nginx-with-built-in-load-balancing-and-caching>

Securing Plone-Sites with https and nginx

For instructions how to use SSL for all authenticated traffic see this blog-post:

- <http://www.starzel.de/blog/securing-plone-sites-with-https-and-nginx>

Setting log files

nginx.conf example:

```
worker_processes 2;
error_log /srv/site/Plone/zinstance/var/log/nginx-error.log warn;

events {
    worker_connections 256;
}

http {
    client_max_body_size 10M;

    access_log /srv/site/Plone/zinstance/var/log/nginx-access.log;
```

Proxy Caching

Nginx can do rudimentary proxy caching. It may be good enough for your needs. Turn on proxy caching by adding to your nginx.conf or a separate conf.d/proxy_cache.conf:

```
##
# common caching setup; use "proxy_cache off;" to override
##
proxy_cache_path /var/www/cache levels=1:2 keys_zone=thecache:100m max_size=4000m
↳inactive=1440m;
proxy_temp_path /tmp;
proxy_redirect off;
proxy_cache thecache;
proxy_set_header Host $host;
proxy_set_header X-Real-IP $remote_addr;
proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
client_max_body_size 0;
client_body_buffer_size 128k;
proxy_send_timeout 120;
proxy_buffer_size 4k;
proxy_buffers 4 32k;
proxy_busy_buffers_size 64k;
proxy_temp_file_write_size 64k;
proxy_connect_timeout 75;
proxy_read_timeout 205;
proxy_cache_bypass $cookie__ac;
proxy_http_version 1.1;
add_header X-Cache-Status $upstream_cache_status;
```

Create a /var/www/cache directory owned by your nginx user (usually www-data).

Limitations:

- Nginx does not support the vary header. That's why we use proxy_cache_bypass to turn off the cache for all authenticated users.
- Nginx does not support the s-maxage cache-control directive. Only max-age. This means that moderate caching will do nothing. However, strong caching works and will cause all your static resources and registry items to be cached. Don't underestimate how valuable that is.

Enabling gzip compression

Enabling gzip compression in Nginx will make your web sites respond much more quickly for your web site users and will reduce the amount of bandwidth used by your web sites.

Instructions for enabling gzip in Nginx:

- <https://varvy.com/pagespeed/enable-compression.html>
- <https://www.nginx.com/resources/admin-guide/compression-and-decompression/>

Production

Description

Hints for Plone in production.

This guide particularly focuses on [Unix-like](#) environments, though the stack discussion may be useful to everyone.

Automatic Plone (re)starts

Introduction

Tips on how to automatically start Plone on server boot.

This manual assumes, that you have installed Plone via the Unified-Installer as `root install`.

If you did a different install, please adjust the examples below to your own needs, the user as which you are running Plone may be different for example.

plonectl script

The general-purpose `plonectl` control command for Plone installations is:

```
yourbuildoutfolder/bin/plonectl
```

`yourbuildoutfolder` is the topmost folder of your Plone installation. It will always contain a `buildout.cfg` file and a `bin` directory.

The `plonectl` command is a convenience script that controls standalone or cluster configurations. In a standalone installation, this will restart the `instance` part. In a ZEO cluster install it will restart the `zeoserver` and `client` parts.

If you have installed Plone in production mode, the Plone server components are meant to be run as a special user, usually `plone_daemon`. (In older versions, this was typically `plone`.) In this case, the `start`, `stop` and `restart` commands are:

```
# start
sudo -u plone_daemon bin/plonectl start
#
# stop
sudo -u plone_daemon bin/plonectl stop
#
# restart
sudo -u plone_daemon bin/plonectl restart
```

Starting on boot

It is best practice to start Plone service if the server is rebooted. This way your site will automatically recover from power loss etc.

On a Linux or BSD system, you have two major alternatives to arrange automatic starting for a production install:

1. A process-control system, like supervisor.
2. Through init.d (BSD rc.d) scripts.

Using supervisor

`supervisor` is a general-purpose process-control system that is well-known and highly recommended in the Plone community.

Process-control systems generally run their controlled programs as subprocesses. This means that the controlled program must not detach itself from the console (daemonize).

Zope/Plone's "start" command does not work for this purpose. Instead use `console`. Do not use `fg` which turns on debug switches that will dramatically slow your site.

Supervisor is well-documented, easy to set up, and included as an installable package with popular Linux and BSD distributions.

Debian LSBInitScripts

Short documentation about how to make an Init Script LSB

This example will start a plone site on boot:

```
#!/bin/sh
### BEGIN INIT INFO
# Provides:          start_plone.sh
# Required-Start:    $remote_fs $syslog
# Required-Stop:     $remote_fs $syslog
# Should-Start:      my plone site
# Default-Start:     2 3 4 5
# Default-Stop:      0 1 6
# Short-Description: Start plone at boot time
# Description:       Start my plone site at boot time
#
#
#
### END INIT INFO

su - plone_daemon -c "/usr/local/Plone/zeocluster/bin/plonectl start"
```

Save this script as `start_plone.sh` in `/etc/init.d` and make it executable.

add the script to dependency-based booting:

```
insserv start_plone.sh
```

Where `start_plone.sh` is an executable init script placed in `/etc/init.d`, `insserv` will produce no output if everything went OK. Examine the error code in `?` if you want to be sure.

This another example (/etc/init.d/plone):

```
#!/bin/sh

### BEGIN INIT INFO
# Provides:          plone
# Required-Start:    $syslog $remote_fs
# Required-Stop:     $syslog $remote_fs
# Should-Start:      $remote_fs
# Should-Stop:       $remote_fs
# Default-Start:     2 3 4 5
# Default-Stop:      0 1 6
# Short-Description: Start plone instances
# Description:       Start the instances located at /srv/Plone/zeocluster/bin/plonectl
### END INIT INFO

PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin

[ -f /usr/local/Plone/zeocluster/bin/plonectl ] || exit 0

DAEMON=/usr/local/Plone/zeocluster/bin/plonectl
NAME="plone "
DESC="daemon zeoserver & client"

. /lib/lsb/init-functions

case "$1" in
    start)
        log_daemon_msg "Starting $DESC" "$NAME"
        if start-stop-daemon --quiet --oknodo --chuid plone:plone \
                               --exec ${DAEMON} --start start
        then
            log_end_msg 0
        else
            log_end_msg 1
        fi
        ;;

    stop)
        log_daemon_msg "Stopping $DESC" "$NAME"
        if start-stop-daemon --quiet --oknodo --chuid plone:plone \
                               --exec ${DAEMON} --start stop
        then
            log_end_msg 0
        else
            log_end_msg 1
        fi
        ;;

    restart)
        log_daemon_msg "Restarting $DESC" "$NAME"
        if start-stop-daemon --quiet --oknodo --chuid plone:plone \
                               --exec ${DAEMON} --start restart
        then
            log_end_msg 0
        else
            log_end_msg 1
        fi
        ;;
esac
```

```
status)
    start-stop-daemon --chuid plone:plone \
                      --exec ${DAEMON} --start status
    ;;

force-reload)
    echo "Plone doesn't support force-reload, use restart instead."
    ;;

*)
    echo "Usage: /etc/init.d/plone {start|stop|status|restart}"
    exit 1
    ;;
esac

exit 0
```

Make sure to read:

<http://wiki.debian.org/LSBInitScripts>

Upstart

Upstart is an event-based replacement for the `/sbin/init` daemon which handles starting of tasks and services during boot, stopping them during shutdown and supervising them while the system is running. It was originally developed for the Ubuntu distribution, but is intended to be suitable for deployment in all Linux distributions as a replacement for the venerable System-V `init`.

Example of a *plone.conf* file in */etc/init/* -> */etc/init/plone.conf*:

```
# Plone - Web-Content Management System
#
# Based on Python and ZOPE

description "start plone"
author "Josh Sehn based on previous work by Christoph Glaubitz"
version "0.3"

console none
respawn

start on (local-filesystems and net-device-up and runlevel [2345])
stop on runlevel [!2345]

exec sudo -u plone_daemon /usr/local/Plone/zeocluster/bin/plonectl start
```

Make sure to read: <http://upstart.ubuntu.com/>

Also check the original source of this sample file: <http://chrigl.de/blogentries/my-plone-configuration>

The above sample has not been extensively tested and is intended for use with in a zeocluster configuration. To use the above sample for a normal (non-root) user installation, replace the last line with:

```
exec /home/${USERID}/Plone/plonectl start
```

Systemd

Create services file *plone.service* in */etc/systemd/system*:

```
[Unit]
Description=Plone content management system
After=network.target

[Service]
Type=forking
ExecStart=/usr/local/Plone/zeocluster/bin/plonectl start
ExecStop=/usr/local/Plone/zeocluster/bin/plonectl stop
ExecReload=/usr/local/Plone/zeocluster/bin/plonectl restart

[Install]
WantedBy=multi-user.target
```

Make systemd take notice of it:

```
systemctl daemon-reload
```

Activate a service immediately:

```
systemctl start plone.service
```

Check status of service:

```
systemctl status plone.service
```

Enable a service to be started on bootup:

```
systemctl enable plone.service
```

More detailed log information:

```
systemd-journalctl -a
```

Make sure to read: <http://www.freedesktop.org/wiki/Software/systemd/>

Crontab

These instructions apply for Debian-based Linuxes.

Example crontab of *yourploneuser*:

```
@reboot /usr/local/Plone/zeocluster/bin/plonectl start
```

rc.local script

For Debian-based Linuxes, add the following line to the */etc/rc.local* script:

```
/usr/local/Plone/zeocluster/bin/plonectl restart
```

Tutorial: Installing Plone for Production on Ubuntu

Description

A step-by-step guide to installing Plone 5.x on a recent Ubuntu LTS [14.04] server installation.

Introduction

This tutorial walks you step-by-step through a minimum responsible installation of Plone for production on a recent Ubuntu LTS server.

The installation includes Plone itself; nginx for a reverse-proxy; a send-only mail-transfer agent; and firewall rules. We'll set Plone to start with server startup and will add cron jobs to periodically pack the database and create snapshot backups.

This minimal install will work for production for a smaller Plone site, and will provide a good base for scaling up for a larger site.

Requirements

1. A clean installation of a recent Ubuntu server. The tutorial has been tested on cloud and virtual box servers. The install described here will run in 512 MB RAM. More RAM will be needed for larger or busy sites.
2. A hostname for the new site. You or your DNS admin should have already created a hostname (e.g., `www.yoursite.com`) and a host record pointing to the new server.
3. Unix command-line and basic system administrator skills. You should know how to use `ssh` to create a terminal session with your new server. You should know how to use `vi` or some other terminal editor.
4. An Internet connection.

Step 1: Platform preparation

Get to the point where you can ssh to the server as a non-root user and use `sudo` to gain root permissions.

First step with any new server is to update the already installed system libraries:

```
sudo apt-get update
sudo apt-get upgrade
```

Then, install the platform's build kit, nginx, and supervisor

```
sudo apt-get install build-essential python-dev libjpeg-dev libxslt-dev supervisor_
↪ nginx
```

Step 2: Install Plone

Please take some time and read the chapter about installation.

Note: Note that this is *root* installation `sudo ./install.sh`. The installer will create special system users to build and run Plone.

Note: This creates a *zeo* installation with two Plone clients. We will only connect one of those clients to the Internet. The other will be reserved for debugging and administrator access. If you know this is a larger site and wish to use load balancing, you may create more clients with the `-clients=##` command-line argument to create more clients. They're also easy to add later.

If you hit an “lxml” error during installation (ie the log shows “Error: Couldn’t install: lxml 2.3.6”) you may need *additional libraries*.

When the install completes, you’ll be shown the preset administrative password. **Record it !** If you lose it, you may see it again:

```
sudo cat /usr/local/Plone/zeocluster/adminPassword.txt
```

Step 3: Set Plone to start with the server

We’re going to use *supervisor* to start Plone with the server. To do so, we’ll create a supervisor configuration file:

```
sudo vi /etc/supervisor/conf.d/plone5.conf
```

Specify that supervisor should start the database server and client1 automatically:

```
[program:plone5server]
user=plone_daemon
directory=/usr/local/Plone/zeocluster
command=/usr/local/Plone/zeocluster/bin/zeoserver fg

[program:plone5client1]
user=plone_daemon
directory=/usr/local/Plone/zeocluster
command=/usr/local/Plone/zeocluster/bin/client1 console
stopwaitseconds=30
```

When that file is saved you’re set to start on server start. To start immediately, tell supervisor about the new components:

```
sudo supervisorctl
supervisor> reread
supervisor> add plone5server
plone5server: added process group
supervisor> add plone5client1
plone5client1: added process group
supervisor> status
plone5client1          RUNNING      pid 32327, uptime 0:00:02
plone5server           RUNNING      pid 32326, uptime 0:00:08
```

Step 4: Create a Plone site

At this point, you should be able to open a web browser and point it to port 8080 on your new server. Do so, and use your administrative password to create a Plone site with the id “Plone”. (Feel free to use a different ID, just remember it below when you set up virtual hosting rules.)

Step 5: Set up virtual hosting

We’re going to use nginx as a reverse proxy. Virtual hosting will be established by rewrite rules. You need two bits of information: 1) the hostname you want to use (for which DNS records should already be set up); 2) the id of the Plone site you created.

We’ll set up nginx by adding a new configuration file:

```
sudo vi /etc/nginx/sites-available/plone5.conf
```

Add the contents

```
server {
    server_name www.yourhostname.com;
    listen 80;

    location / {
        rewrite ^/(.*)$ /VirtualHostBase/http/www.yourhostname.com:80/Plone/
↪VirtualHostRoot/$1 break;
        proxy_pass http://localhost:8080;
    }
    location ~* manage_ {
        deny all;
    }
}

server {
    server_name yourhostname.com;
    listen 80;
    access_log off;
    rewrite ^/(.*)$ http://www.yourhostname.com$1 permanent;
}
```

And save.

Note: The *location ~* manage_* rule will deny access to most of the Management interface. (You’ll get to that by bypassing nginx.)

Note: The second server stanza sets up an automatic redirect that will transfer requests for the bare hostname to its *www.* form. You may not want or need that.

Enable the new nginx site configuration:

```
cd /etc/nginx/sites-enabled
sudo ln -s /etc/nginx/sites-available/plone5.conf
```

And, tell nginx to reload the configuration:

```
sudo service nginx configtest
sudo service nginx reload
```

Try out your virtual hosting.

Step 6: Set up packing and backup

We want the Zope database to be packed weekly. We'll do so by setting up a *cron* job:

```
sudo vi /etc/cron.d/zeopack
```

Add the contents:

```
57 22 * * 5 plone_daemon /usr/local/Plone/zeocluster/bin/zeopack
```

And save.

Note: Pick a time when your system can take some extra load. Don't use the day/time above.

Let's also create a daily snapshot of the database:

```
sudo vi /etc/cron.d/plonebackup
```

Add the contents below, adjust the time, and save:

```
37 0 * * * plone_daemon /usr/local/Plone/zeocluster/bin/snapshotbackup
```

Note: This snapshot will give you a stable copy of the database at a particular time. You'll need a separate strategy to backup the server's file system, including the snapshot.

Step 7: Add a send-only Mail Transfer Agent

You don't need this step if you have an MTA on another server, or are using a mail-send service. If you don't have that available, this step will create a localhost, port 25, MTA that you may use with Plone's mail setup.

We're going to use Postfix. There are lots of alternatives.

Add the Postfix package and edit its main configuration file:

```
sudo apt-get install postfix
sudo vi /etc/postfix/main.cf
```

Change the bottom section to turn off general mail in:

```
myhostname = www.yourhostname.com
alias_maps = hash:/etc/aliases
alias_database = hash:/etc/aliases
myorigin = yourhostname.com
mydestination =
relayhost =
mynetworks = 127.0.0.0/8 [::ffff:127.0.0.0]/104 [::1]/128
mailbox_size_limit = 0
```

```
recipient_delimiter = +
inet_interfaces = loopback-only
```

Tell postfix to restart:

```
sudo /etc/init.d/postfix restart
```

Step 8: Set up a firewall

You *must* set up a firewall. But, you may be handling that outside the system, for example via AWS security groups.

If you want to use a software firewall on the machine, you may use *ufw* to simplify rule setup.

```
sudo apt-get install ufw
sudo ufw limit 22/tcp
sudo ufw allow 80/tcp
sudo ufw allow 443/tcp
sudo ufw enable
```

Note: This blocks everything but SSH and HTTP(S).

You may be wondering, how do you do Management Interface administration? SSH port forwarding will allow you to build a temporary encrypted tunnel from your workstation to the server.

Execute on your workstation the command:

```
ssh yourloginid@www.yourhostname.com -L:8080:localhost:8080
```

Now, ask for <http://localhost:8080/> in your workstation web browser, and you'll be looking at the Management Interface.

Scaling up

This installation will do well on a minimum server configuration (512MB RAM). If you've a larger site, buy more memory and set up reverse-proxy caching and load balancing.

Deploying and installing Plone in production is a good introduction to scaling topics.

Plone Security Best Practices

Description

These are security recommendations that should be used in production environments and in exposed staging and developments environments.

No software is perfect, and new attacks are constantly being developed as technology and browsers evolve. For security reasons, you should always follow 'defensive tactics', protecting your sites as well as possible against even unknown attacks.

If you follow these security best practices for production environments and for exposed staging and development environments, you ensure that you have done everything possible to secure your systems.

Where to obtain and provide security information

Mailing list: plone-announce@lists.sourceforge.net

Security announcements as well as new Plone releases will be published on the [plone-announce mailing list](#)

Ensure that you are subscribed to it if you manage a Plone installation.

Whom to inform on potential security vulnerabilities or attacks

If you have found a potential security vulnerability in Plone or in any component of a normal setup stack, please inform the Plone Security Team at security@plone.org. If necessary, the Plone Security Team will in turn inform and work with the security teams of other software packages.

Please do not disclose unfixed security vulnerabilities by sending information or exploits via public mailing lists or on public issue trackers.

If you have any questions, please do not hesitate to contact the Plone Security Team at security@plone.org

If you suspect that your Plone site is undergoing an attack, also feel free to contact the Plone Security Team at security@plone.org. It helps the team to see if a new vulnerability or attack vector is used, and the team could provide you with advice on how to recover from and prevent this attack.

Be prepared

Note: Many of these precautions are valid for any software stack, not just Plone.

Monitor your systems

Security is also about staying alert to unusual events. Use a monitoring system that checks for unusual access patterns or for a sudden increase in website traffic. Either scenario should be reason for concern.

Backups & Recovery tests

Always make sure you keep regular backups of your site content, and keep those backups in a safe place. It is equally important to check that you are able to restore such a backup, and to know the procedure to bring a site back online. You should test your backup and restore procedures by using them before you run into an emergency!

These are standard security procedures for any software stack, not just Plone.

Keep your Plone up-to-date

Always keep your software up-to-date. Current versions will be patched and you have a smoother upgrade path.

Work with the lowest amount of privileges necessary

Like on your OS, always work as a user-account with the lowest amount of privileges necessary to get the current job done. After doing management work, log out, and log back in as your ‘normal user’.

Use dedicated Unix user accounts

Create these two Unix user accounts:

- *plone_buildout*
- *plone_daemon*

Both accounts should be in the same group *plone_group*

The *plone_buildout* user should be used for buildout and precompile recipes.

The *plone_daemon* user that actually runs your Zope/Plone instance should not be able to modify the source code of your project.

The *plone_buildout* user needs read and write access to your whole buildout directory and to buildout_cache. The *plone_daemon* user only needs to write into your var directory.

Plone's Unified Installer uses this scheme.

“Zope Manager” User should not be used for daily work

Zope is an application server that provides the foundation for your Plone instance. It has done a lot about application security and provides a fine grained permission setting. But nowadays in the context of Plone the Management Interface is only used by users who have a “Zope Manager” role, which gives full access to everything in Zope.

The Management Interface was created before JavaScript exploits were common, so it did not implement security against exploits like CSRF.

You should use a user with Zope Manager role only if the permissions are really needed, such as when initially creating a Plone site.

While using the Management Interface, we recommend disabling JavaScript in your browser.

Use dedicated Plone ‘Site Administrator’ users

The first step on a new created Plone site should be to create a dedicated user with ‘Site Administrator’ role. This user should only be used for maintenance and administrative purposes. Your daily content-editing account should not have extra privileges.

Lock down access to your Management components

Firewall

Use a firewall to restrict access to only the HTTP (80) and/or HTTPS(443) ports, so that the Zope client(s) are not directly accessible via their own port(s).

Access control

Your Management Interface should not be available via the production domain. The following rules will block all common Management Interface pages

For Apache httpd (2.2 syntax)

```
RewriteRule ^(.*)manage(_.*)$ - [L,NC]
<LocationMatch "^/(manage|manage_main|(.*)/manage(_.*))$" >
    Order deny,allow
    Deny from all
</LocationMatch>
```

For nginx:

```
location ~* /manage(_.*)?$ {
    return 403;
}
```

set HTTP Security Headers

Always use as strict security headers as possible:

```
Header set X-Frame-Options "SAMEORIGIN"
Header set Strict-Transport-Security "max-age=15768000; includeSubDomains"
Header set X-XSS-Protection "1; mode=block"
Header set X-Content-Type-Options "nosniff"
# Header set Content-Security-Policy-Report-Only "default-src 'self'; img-src *;
↪style-src 'unsafe-inline'; script-src 'unsafe-inline' 'unsafe-eval'"
Header set Content-Security-Policy "default-src 'self' cdn.example.com www.example.
↪com; \
script-src 'self' 'unsafe-inline' 'unsafe-eval' cdn.example.com www.example.com; \
style-src 'self' 'unsafe-inline' cdn.example.com www.example.com *.example.com; \
img-src 'self' 'unsafe-inline' cdn.example.com www.example.com *.example.com; \
font-src 'self' 'unsafe-inline' cdn.example.com www.example.com *.example.com; \
object-src 'self' cdn.example.com www.example.com *.example.com;
```

Use caution when using SSH tunnels to access Management Interface

Once you have stripped down access to your Management Interface via your normal domain URLs, take care you don't accidentally bypass the security by allowing CSRF hijacking via an SSH tunnel. <http://127.0.0.1:8080/> and <http://localhost:8080/> are common attack vectors via JavaScript. Make sure you close all other browser tabs (or open a different browser, e.g. Firefox when you normally use Safari) when accessing these URL's. Always close the SSH tunnel after you are done with maintenance. Alternatively consider using a dedicated manage domain.

Provide a dedicated manage domain

Apache Example

```
<VirtualHost *:443>

    ServerAdmin webmaster@example.com
    ServerName manage@example.com

    SSLEngine on
```

```
# Only use TLS 1.0+ no old SSLv2 or SSLv3
SSLProtocol all -SSLv2 -SSLv3

# Limit Cipher algorithm to strong ones, openssl ciphers 'HIGH:!MEDIUM:!aNULL:!
↪MD5:-RSA' should show those
SSLCipherSuite HIGH:!MEDIUM:!aNULL:!MD5:-RSA

# Certificate
SSLCertificateFile manage.example.com.pem
# Private Key
SSLCertificateKeyFile manage.example.com_key.pem

# Certificate Chain of applicable
SSLCertificateChainFile example.com.crt

ProxyVia On
ProxyRequests Off
ProxyPreserveHost On
# prevent your web server from being used as global HTTP proxy
<LocationMatch "[^/]">
    Deny from all
</LocationMatch>

<Proxy *>
    Order deny,allow
    Allow from all
</Proxy>

<Location />
    Order Deny,Allow
    Deny from All
    Allow from IP-Zone # Control your IP Zone to Access
    AuthType # Use a separate Authentication Protocol

</Location>

Header set X-Frame-Options "SAMEORIGIN"
Header set Strict-Transport-Security "max-age=15768000; includeSubDomains"
Header set X-XSS-Protection "1; mode=block"
Header set X-Content-Type-Options "nosniff"
Header set Content-Security-Policy "default-src 'self' cdn.example.com www.
↪example.com; \
    script-src 'self' 'unsafe-inline' 'unsafe-eval' manage.example.com; \
    style-src 'self' 'unsafe-inline' manage.example.com *.example.com; \
    img-src 'self' 'unsafe-inline' manage.example.com; \
    font-src 'self' 'unsafe-inline' manage.example.com; \
    object-src 'self' manage.example.com;

# You could manage all included Controls via this one channel
```

```
# Example for HAProxy
ProxyPass /haproxy-status http://127.0.0.1:8000/haproxy-status
ProxyPassReverse /haproxy-status http://127.0.0.1:8000/haproxy-status

# Rewrite for Zope Root
RewriteRule ^/(.*)$ http://127.0.0.1:8080VirtualHostBase/https/manage.example.
↪com:443/VirtualHostRoot/$1 [P,L]

</VirtualHost>
```

Status check

Introduction

It is good practice to regularly check if your web site is still up. This may be done by a command line tool like `httpok` or some online service. For this you don't want to query a possibly expensive page. And you don't want to query anything that may be stored in an intermediate caching server. There are a few options.

ok browser view

Call `http://plonesite-url/@@ok`. Or if a tool is confused by the `@` signs, call `http://plonesite-url/ok`. You can also call this on the Zope site root. This returns the text `OK` and sets headers to avoid caching.

This was introduced in the `Products.CMFPlone` package in Plone 4.3.12, 5.0.7, and 5.1b1.

ZopeTime

Call `http://plonesite-url/ZopeTime`. You can also call this on the Zope site root. This will return the current time on the server.

This may be served by a caching server in front of Plone, so it does not guarantee that Plone is still live.

If you use the `experimental.publishtraverse` package, this will either give you a `NotFound` error, or log a warning each time it is called, due to lacking permissions. And this may happen in core Plone in the future too.

Plone Upgrade Guide

Description

Instructions and tips for upgrading to a newer Plone version.

This guide particularly focuses on [Unix-like](#) environments, though the stack discussion may be useful to everyone.

Upgrading Plone

This document covers the procedures and issues involved in upgrading an existing Plone installation.

This involves both the upgrading of the program set, and migration of the site itself.

Generally, you will often see the word *migration* used as the word we use to describe the process of getting your Plone site from one version of a given component to a newer version.

For most people, this means upgrading Plone to a newer release, for example from 5.0.x to 5.1.x.

Migration is necessary because the internals of Plone sometimes change to support new functionality. When that's the case, the content which is stored in your Plone instance may not match what the new version of the software expects.

Plone has a builtin tool that migrates existing content to the new structure.

This guide describes migration in Plone, specifically how you upgrade between different versions.

Before migrating you should read this entire document to understand the potential impact migrating will have on your Plone site.

It is also wise to have read the [troubleshooting](#) section, in case you may need to employ one of the techniques there.

The guide applies to all contemporary versions of Plone.

For unsupported versions from the year 2009 and before, see older versions of this documentation.

Version Numbering And Terminology

Plone has a policy that increases the version number to a .0 on every major release.

This means that when we say a *major release*, we are referring to a x.0 release, whereas a minor release has the version numbering 4.3.14 or 5.1.0

In addition to the general procedure there are *version-specific migration guides*.

These guides contain more specific instructions and valuable information that has been collected from real-life migration cases.

No Large Leaps

Note:

- It is advisable to not make large leaps in version numbers.
 - A single upgrade should not try to bridge multiple major version numbers.
-

Going from Plone 4.0 to Plone 5.1 is fine.

If you are at Plone 2.5 and want to upgrade to the latest Plone 5, you should approach this in several steps:

- First upgrade from Plone 2.5 to the latest Plone 3 version (3.3.6).
- Then upgrade from Plone 3 to the latest Plone 4 version.
- Then upgrade from Plone 4 to the latest Plone 5 version.

Preparations

Description

Things to do before you migrate Plone.

Gather information

- Read the “What’s new in...” for your relevant Plone version, and read the release notes.
 - You’ll find these in the CMFPlone directory of the distribution of the new version of Plone.
- Make sure to check installed add-ons, if you **do not** need certain add-ons anymore, please deactivate and deinstall.
- Check for dependencies
 - Read the release notes for the Plone release you are upgrading to, in particular:
 - What version of Python is required?
 - What version of Zope is required?
 - Do you need any new python libraries?
 - Make sure all the add-on products you are using have updated to support the version of Plone you are upgrading to.
 - Start with the third-party products that are in use on your site.
 - * Verify that they have been updated or verified to work on the new version, and get them upgraded in your existing instance before you start the Plone/Zope/Python upgrade if possible.
 - If Zope depends on a newer version of Python, install the new version of Python first.
 - If the newer version of Plone depends on a newer version of Zope, you will need to install that before proceeding with the Plone upgrade.

Note: Zope has its own migration guidelines, which you will find in the release notes of the version you are migrating to.

If Plone is being upgraded at the same time as a Zope version, Plone will usually handle the Zope upgrade with its own migration script.

- Read the following files in the CMFPlone directory of the distribution of the new version of Plone you want to update to:
 - README.txt
 - INSTALL.txt
 - UPGRADE.txt (although this usually contains only the general procedure outlined above)
 - * These files are important because they may contain important last minute information and might be more specific than the relevant sections of this reference manual.

Back up your Plone site

Note: It’s important to back up your Plone site.

You will find a *[how-to on backing up your Plone site here](#)*.

Setup a test environment to rehearse the upgrade

Note: Never work directly on your live site until you know that the upgrade was successful.

Instead, create a test environment to rehearse the upgrade. Copy your instance into a new environment and upgrade the copy. This is a good way of working out your third party products and dependencies in preparation for the final upgrade of the live site!

Upgrade add-on products

Description

The steps to take to migrate your third party products

- Shut down your Plone server instance.
- If you specified concrete versions of the third-party products in your *buildout.cfg* file (so-called “pinning”, and a recommended practice), like *Products.PloneFormGen = 1.7.17*, update these references to point to the new versions.

Note: Without pinning, i.e. specifying only, for example, *Products.PloneFormGen* and no version, buildout will pick the newest version of the products by default.

- Run *bin/buildout*. Wait until all new software is downloaded and installed.
- Start Plone again - your site may look weird, or even be inaccessible until you have performed the next step
- **Navigate to the Add-on screen (add `/prefs_install_products_form` to your site URL, and upgrade products if you have any)**
 - Perform product-specific upgrade procedures (if any).
 - You will find these in the documentation of each product.

Should the `/prefs_install_products_form` be unreachable, you should try doing the add-on upgrades from the Management Interface. Navigate to the quickinstaller in the Management Interface, and reinstall or upgrade products that are shown to be outdated.

Note: Be careful when updating add-ons through the Management Interface. It may show outdated themes as well with a hint to update. If you do that, the updated theme will activate itself, overriding your current theme. If this happens, re-enable your theme in the theming panel.

Troubleshooting

Description

What to do when a problem occurs during a Plone upgrade.

When a problem occurs during the migration we recommend that you take the following steps.

Check the log files

When a site error occurs, or Zope fails to start, there's probably an informative error message in Zope's log files. Locate [these log files](#) and inspect instance.log. Ignore irrelevant warnings and search for words such as error, exception and traceback (case-insensitive).

When Zope doesn't start and there's no useful information in the log file, you can start Zope interactively and watch for error messages in the output::

```
bin/instance fg
```

You may be able to find more information on the error messages in:

- the *Version-specific migration tips* for your version of Plone
- the *Error References*

Test without customizations

When you have customized page templates or Python scripts, your changes may interfere with changes in the new version of Plone. It's important to rule out this possibility, since your customizations are unique to your site and no one on the planet will be able to help you solve it.

Temporarily remove your customizations, for example by removing your layers from portal_skins, or by removing files from these layers on the file system. If the problem disappears, you'll need to double-check your customizations. It's usually best to copy the original files of the new version of Plone to your skin, and re-customize those.

Test without products

Bugs or compatibility problems in products that you have installed may cause problems in Plone. Go to Site Setup > Add/Remove Products and remove (uninstall) all product that are not distributed with Plone. Remove the uninstalled products from the Products directory of your Zope instance.

If the problem disappears, you'll need to doublecheck the offending product:

- Does it support the new version of Plone, Zope and Python? Check the product's README.txt or other informational files or pages.
- Does the product require any additional migration procedures? Check the product's INSTALL.txt, UP-GRADE.txt or other informational files or pages.
- Does the product install properly? Re-install it and check the install log.

Test with a fresh Plone instance

Create a new Plone site with your new version of Plone. You don't need a new Zope instance, since you can add another Plone site in the root of Zope. If the problem does not occur in a fresh site, the cause of your problem is most likely a customization, an installed product or content that was not migrated properly.

Make the problem reproducible

Before you go out and [ask for help](#), you should be able to describe your problem in such a way that others can reproduce it in their environment.

Reduce the problem to the smallest possible domain. Eliminate products and customizations that are not part of the problem. This makes it easier for others to reproduce the problem and it increases your chances of meeting others

with the same problem or even a solution. The more complex your story is, the more likely that it is unique to your situation and in-penetrable to others.

Ask for help on a mailing list

Ask for help on the [Plone setup list](#). Be sure to:

- Provide relevant source code for your customizations that are part of the problem.
- Describe the exact configuration, software versions, migration history, error messages and so on.

Report a bug

Once you have investigated, analyzed, identified and confirmed the cause of your problem and you are convinced it's a bug (rather than an X-file), go to the appropriate bug tracker and report it:

- Products: the README usually tells how to report bugs
- [Plone Issue Tracker](#)

Do not use the bug trackers to ask for help. First analyze your problem and assert that it's a bug before you report it.

Version-specific migration procedures and tips

Description

In addition to the general procedure described in the previous sections, this section provides version-specific procedures and tips. If your migration does not involve a version pair specified here, then you may follow the general procedures alone.

Upgrading Plone 5 within 5.x.x series dot minor releases

Description

Steps for minor upgrades within the Plone 5 Major Release.

Warning: Before performing any Plone upgrade, you should always have a complete backup of your site. See the [Preparations](#) section of this manual for more details.

In addition, you should check the [Version-specific migration tips](#) section of this manual for any notes that may apply to the specific version upgrade you're about to perform.

Buildout

Out of the box, Plone's Unified Installer includes a `buildout.cfg` (typically located at `your-plone-directory/zinstance/buildout.cfg`) file that contains the following parameter.

```
extends =
base.cfg
versions.cfg
# http://dist.plone.org/release/5.1-latest/versions.cfg
```

This tells buildout to get all of its package versions from the included versions.cfg file. Notice that there is another line, commented out, that points to dist.plone.org. This location will always contain the most recent versions that comprise the latest release in the Plone 5.1 series. (You can also replace 5.1-latest with 5.0-latest or 5.2-latest, or another other existing minor release in the 5.x series.)

To upgrade your buildout to use the latest Plone 5.1.x release, comment out versions.cfg and uncomment the line pointing to dist.plone.org, so it looks like this:

```
extends =
base.cfg
# versions.cfg
http://dist.plone.org/release/5.3-latest/versions.cfg
```

Save your changes.

Upgrading

Stop your Plone instance:

```
bin/plonectl stop
```

Rerun buildout:

```
bin/buildout
```

This may take a some time, as Plone downloads new releases.

When buildout finishes running, restart your Plone instance:

```
bin/plonectl start
```

Migration

Visit your Management Interface (<http://yoursite:8080>). You will see a message prompting you to run Plone's migration script for each site in your instance:

This site configuration is outdated and needs to be upgraded.

Click Upgrade button next to the site and the upgrade will run.

Check the *Dry Run* checkbox if you want to test the migration before you execute it.

Upgrading Plone 4.x to 5.0

Description

Instructions and tips for upgrading to a newer Plone version.

Note: If you want to upgrade add-ons to Plone 5, also see: [../../develop/addons/upgrade_to_50.rst](#)

General information

- Before you upgrade read [../into.rst](#) and [../preparations.rst](#).
- Always upgrade from the latest version of 4.x to the latest version of 5.x (at the time of writing 4.3.7 to 5.0.2). This will resolve many migration-specific issues.
- If you have problems don't be afraid to ask for help on <http://community.plone.org>
- There is a video of a talk “How to upgrade sites to Plone 5”: <https://youtu.be/bQ-IpO-7F00?t=1m17s> (slides: <http://de.slideshare.net/derschmock/upgrade-to-plone-5>)

Changes due to implemented PLIPS

PLIPs are Plone Improvement Proposals. These are about larger changes to Plone, discussed beforehand by the community.

PLIP 13350 “Define extra member properties TTW”

In Plone 5, the custom fields displayed in the user profile form and the registration form are managed by *plone.schemaeditor*. They are dynamically editable from the Plone control panel, and can be imported from a Generic Setup profile file named *userschema.xml*.

If you have some custom member properties in your Plone site, be aware that:

- extra attributes defined in *memberdata_properties.xml* will be preserved, but they will not be automatically shown in the user profile form or the registration form,
- if you have implemented some custom forms in order to display your custom member attributes, they will not work anymore as *plone.app.users* is now based on *z3c.form*. You can replace them by declaring their schema in *userschema.xml*.

Note: When a custom field is defined in *userschema.xml*, its corresponding attribute is automatically created in the *portal_memberdata* tool, so there is no need to declare it in *memberdata_properties.xml*. *memberdata_properties.xml* will only handle attributes that are not related to the user profile form or the registration form.

PLIP 13419 “Improvements for user ids and login names”

Since Plone 4.0 you could switch to using email as login in the security control panel. In Plone 5.0 some related improvements were made. When you are already using email as login, during the Plone 5.0 migration the login names will be transformed to lowercase. When the email addresses are not unique, for example you have both `joe@example.org` and `JOE@example.org`, the migration will *fail*.

Best is to fix this in your site in Plone 4, by changing email addresses or removing no longer needed users. When there are only a few users, you can do this manually. To assist you in sites with many users, in Plone 4.1 and higher, you can add the `collective.emaillogin4` package to the eggs of your Plone instance. With that package, even without installing it in the add-ons control panel, you can call the `@@migrate-to-emaillogin` page to look for duplicate email addresses.

Note: This PLIP basically integrates the `collective.emaillogin4` package in Plone 5.

Other PLIP changes

PLIPs that resulted in changes that will have to be documented in this upgrade-guide.

plone.api Todo: Tell people to use it. Explain how to configure `plone.recipe.codeanalysis` to check for old-style code
Roel

plone.app.multilingual Todo: How to migrate from LP to PAM

Convert control panels to use z3c.form Todo: How to migrate your custom control-panels Who: Tisto

Main_template rebirth to HTML5 bloodbare Todo: What to do when you customized your main_templates. Who:
?

Automatic CSRF Protection Todo: How to protect your existing forms Who: Nathan

Linkintegrity in Plone 5 Who: pbauer

security setting changes

complex, look `betas.py` **TODO**

mail setting changes

markup setting changes

user_and_group setting changes

social media changes

imaging changes

login setting changes

Changed imports and functions

Products.CMFPlone.interfaces.IFactoryTool

This is now moved to `ATContentTypes`.

Example:

```
try:
    # Plone 4
    from Products.CMFPlone.interfaces import IFactoryTool
except ImportError:
    # Plone 5
    from Products.ATContentTypes.interfaces.factory import IFactoryTool
```

plone.app.multilingual

Note: The preferred translation add-on for Plone 5 is plone.app.multilingual. This package supersedes LinguaPlone.

Warning: This is still work in progress

There are 3 different parts to the migration from LinguaPlone to plone.app.multilingual:

- From LP to PAM 2.X - on Plone 4 and then to Plone 5 (PAM 3.X)
See: <https://github.com/plone/plone.app.multilingual/issues/181>
- From PAM 1.X to 2.X - on Plone 4 and then to Plone 5 (PAM 3.X)
Step 1: plone.multilingual is merged into plone.app.multilingual. Imports in your custom code needs to be changed: See: <https://github.com/plone/plone.app.multilingual/issues/181#issuecomment-141661848>
Step 2: Removed plone.multilingualbehavior: <https://github.com/plone/plone.app.multilingual/issues/183>
Step 3: TODO
- From PAM 2.X on Plone 4 to Plone 5 (PAM 3.X)
Step 1: plone.multilingual is merged into plone.app.multilingual. Imports in your custom code needs to be changed: See: <https://github.com/plone/plone.app.multilingual/issues/181#issuecomment-141661848>
<https://github.com/plone/Products.CMFPlone/issues/1187>

Archetypes

Plone 5 now uses dexterity as the content type engine instead of Archetypes.

For packages that still use Archetypes, you'll need to install the ATContentTypes base package.

The easiest way to get the dependencies for Archetypes (uuid_catalog, reference_catalog, archetypes_tool) is to add the following profile to your dependencies in `metadata.xml`:

```
<dependencies>
...
  <dependency>Products.ATContentTypes:base</dependency>
</dependencies>
```

See <https://github.com/smcMahon/Products.PloneFormGen/blob/master/Products/PloneFormGen/profiles/default/metadata.xml> for a working example.

Resource Registry

See also:

<http://docs.plone.org/adapt-and-extend/theming/resourceregistry.html>

Plone 5 introduces some new concepts, for some, with working with JavaScript in Plone. Plone 5 utilizes Asynchronous Module Definition (AMD) with `requirejs`. We chose AMD over other module loading implementations (like `commonjs`) because AMD can be used in non-compiled form in the browser. This way, someone can click “development mode” in the resource registry control panel and work with the non-compiled JavaScript files directly.

Getting back on point, much of Plone’s JavaScript was or still is using JavaScript in a non-AMD form. Scripts that expect JavaScript dependency scripts and objects to be globally available and not loaded synchronously will have a difficult time figuring out what is going on when upgrading to Plone 5.

There are two scenarios where this will happen that we’ll tackle in this post. 1) You have JavaScript registered in `portal_javascripts` that are not AMD compatible. 2) You have JavaScript included in the head tag of your theme and/or specific page templates that are not AMD compatible.

1) Working with deprecated `portal_javascripts`

The deprecated resource registries (and `portal_javascripts`) has no concept of dependency management. It simply allowed you to specify an order in which JavaScript files should be included on your site. It also would combined and minify them for you in deployment mode.

Registration changes

Prior to Plone 5, JavaScript files were added to the registry by using a [Generic Setup Profile](#) and including a `jsregistry.xml` file to it. This would add your JavaScript to the registry, with some options and potentially set ordering.

In Plone 5.0, Plone will still recognize these `jsregistry.xml` files. Plone tries to provide a shim for them. It does this by adding all `jsregistry.xml` JavaScripts into the “plone-legacy” Resource Registry bundle. This bundle includes a global jQuery object and includes the resources in sequential order after it.

However, you should consider at least migrating your resources as described in <https://github.com/collective/example.p4p5> to gain control over your dependencies or if you want to keep backward compatibility to older Plone versions in your Add-ons.

Old style `jsregistry.xml`

An old style Resource Registry would look like this:

```
<?xml version="1.0"?>
<object name="portal_javascripts">
  <javascript
    id="++resource++foobar.js"
    inline="False"
  />
</object>
```

To migrate this to Plone 5, resource registrations are all done in the [Configuration Registry](#).

New style with `registry.xml`

The new registration will look something like:

```
<?xml version="1.0"?>
<registry>
  <records prefix="plone.resources/foobar"
    interface='Products.CMFFlone.interfaces.IResourceRegistry'>
    <value key="js">++resource++foobar.js</value>
    <value key="deps">jquery</value>
  </records>
</registry>
```

Notice how I’ve now added the `deps` property of “jquery”. This is not necessary – I’m just giving an example that this script needs a global jQuery available.

This alone will not get your JavaScript included however. In order to modernize our JavaScript stack, Plone needed to make some changes with how it included JavaScript. All we’ve done so far is define a resource. In order for a resource to be included, it needs to be part of a bundle. A bundle defines a set of resources that should be compiled together and distributed to the browser.

You either need to add your resource to an existing bundle or create your own bundle.

In this post, we’ll describe the process of creating your own bundle. Again, we use `registry.xml` for configuration:

```
<records prefix="plone.bundles/foobar"
  interface='Products.CMFPlone.interfaces.IBundleRegistry'>
  <value key="resources">
    <element>foobar</element>
  </value>
  <value key="enabled">True</value>
  <value key="jscompilation">++resource++foobar-compiled.min.js</value>
  <value key="last_compilation">2015-02-06 00:00:00</value>
</records>
```

One important aspect here is the “jscompilation” settings. This defines the compiled resource used in production mode.

But, it’s a bit more work

Yes, we know. We tried very hard to figure out the easiest way to modernize Plone’s JavaScript development stack. The old, sequential inclusion is not useful these days.

That being said, adding resources, bundles and compiling them can all be done Through The Web(TTW) in the new Resource Registries configuration panel. That way you can turn on development mode, compile your resources and then copy that compiled version into your package for distribution and not need to know any newfangled nodejs technologies like grunt, gulp, bower, npm, etc.

Updating non-AMD scripts

If you are not including your JavaScript in the Resource Registries and just need it to work alongside Plone’s JavaScript because you’re manually including the JavaScript files in one way or another(page templates, themes), there are a number of techniques available to read on the web that describe how to make your scripts conditionally work with AMD.

For the sake of this post, I will describe one technique used in Plone core to fix the JavaScript. The change we’ll be investigating can be seen with [in a commit to plone.app.registry](#). `plone.app.registry` has a control panel that allows some Ajax searching and modals for editing settings.

To utilize the dependency management that AMD provides and have the JavaScript depend on jQuery, we can wrap the script in an AMD *require* function. This function allows you to define a set of dependencies and a function that takes as arguments, those dependencies you defined. After the dependencies are loaded, the function you defined is called.

Example:

```
require([
  'jquery',
  'pat-registry'
], function($, Registry) {
```



```
'use strict';
...
// All my previous JavaScript file code here
...
});
```

Here, the two dependencies we have are jQuery and the pattern registry. I will not get into the pattern registry as it's off topic for this discussion—it is basically a registry of JavaScript components. The necessity for using it here is with Ajax calls and binding new DOM elements dynamically added to the page.

Additionally, above this *require* call, I provide some backward compatible code that you can inspect. It's not necessary in this case but I added it to show how someone could make their script work when *requirejs* was available and when it was not.

Caveats

Compilation

Prior to Plone 5, when a resource was changed or added to the JavaScript registry, the registry would automatically re-compile all your JavaScript files.

In switching to AMD, the compile step is much more resource intensive. It takes so long, there is no way we could do this real-time. Additionally, it can not be done in Python.

When changes are made to existing bundles, re-compilation will need to be done TTW in the Resource Registries control panel. There is a build button next to each bundle. For advanced users, compilation can be done using a tool like grunt in your development environment.

Conditional resources

In Plone 5, individual resources can not be registered conditionally to certain page. This is due to the way we build JavaScript with AMD.

Instead we have Python helper-methods in the Resource Registry to add custom JS and CSS to your views or forms.

Instead of using the legacy fill-slot like this (Plone 4):

```
<metal:slot fill-slot="javascript_head_slot">
...
</metal:slot>
<metal:slot fill-slot="css_slot">
...
</metal:slot>
```

In Plone 5 it's recommended to instead use the new Python methods you can find in `Products.CMFPlone.resources`:

```
from Products.CMFPlone.resources import add_bundle_on_request
from Products.CMFPlone.resources import add_resource_on_request

add_resource_on_request(self.request, 'jquery.recurrenceinput')
add_bundle_on_request(self.request, 'thememapper')
```

This is better than always loading a resource or bundle for your whole site.

Only bundles can be conditionally included. If you have a resource that needs to be conditionally included, it will likely need its own bundle.

Control Panel

In Plone 4.x, the Plone configuration settings have been stored as portal properties spread across the Management Interface. In Plone 5, those settings are all stored as `plone.app.registry` entries in `registry.xml`.

There are now sections in the control panel, this can be set from the `controlpanel.xml`. See the current definitions for more information.

The display of icons for control panels is now controlled by css. The name of the control panel is normalized into a css class, which is applied to the link in the main layout of all control panels. For example, if the “`appId`” of your control panel (as set in `controlpanel.xml` in your install profile) is “`MyPackage`” then the css class that will be generated is “`.icon-controlpanel-MyPackage`”. In order to have an icon for your control panel you must make sure that a css rule exists for that generated css class. An example might be:

```
.icon-controlpanel-MyPackage:before { content: '\e844'; }
```

The value you use for this css rule should identify one of the fontello icons included in Plone, or a font-based icon provided by your package itself.

It is not possible at this time to set an icon for your add-on package control panels without including css in your package.

For documentation on how to use it in your own add-ons see <http://training.plone.org/5/registry.html>

Properties

In the past editor settings were part of the portal properties which contained a site properties object with the relevant attributes.

site properties allowed direct attribute access, so you could access the `available_editors` via:

```
ptools.site_properties.available_editors
```

Now you can access the property via `get_registry_record()`:

```
>>> from plone import api
>>> api.portal.get_registry_record('plone.available_editors')
```

The keys mostly the same, they are only prefixed with *plone*. now. Normally, you do not modify or access these records. Instead you change the settings in your genericsetup profile in the file *propertiestool.xml*

Old Property Sheet	Old Key	New Property
navtree_properties	sortAttribute	TBD
navtree_properties	sortOrder	TBD
navtree_properties	sitemapDepth	TBD
navtree_properties	root	TBD
navtree_properties	currentFolderOnlyInNavtree	TBD
navtree_properties	includeTop	TBD
navtree_properties	topLevel	TBD
navtree_properties	bottomLevel	TBD
navtree_properties	showAllParents	TBD

Continued on next page

Table 5.1 – continued from previous page

navtree_properties	idsNotToList	TBD
navtree_properties	parentMetaTypesNotToQuery	TBD
navtree_properties	metaTypesNotToList	TBD
navtree_properties	enable_wf_state_filtering	TBD
navtree_properties	wf_states_to_show	TBD
site_properties	allowAnonymousViewAbout	plone.allow_anon_views_about
site_properties	displayPublicationDateInByline	plone.display_publication_date_byline
site_properties	default_language	plone.default_language
site_properties	default_charset	TBD
site_properties	ext_editor	plone.ext_editor
site_properties	available_editors	plone.available_editors
site_properties	default_editor	plone.default_editor
site_properties	allowRolesToAddKeywords	TBD
site_properties	autho_cookie_length	plone.auth_cookie_length
site_properties	calendar_starting_year	TBD
site_properties	calender_future_years_available	TBD
site_properties	invalid_ids	TBD
site_properties	default_page	TBD
site_properties	search_results_description_length	plone.search_results_description_length
site_properties	ellipsis	TBD
site_properties	typesLinkToFolderContentsInFC	TBD
site_properties	visible_ids	TBD
site_properties	exposeDCMetaTags	plone.exposeDCMetaTags
site_properties	types_not_searched	plone.types_not_searched
site_properties	search_review_state_for_anon	REMOVED
site_properties	search_enable_description_search	REMOVED
site_properties	search_enable_sort_on	REMOVED
site_properties	search_enable_batch_size	REMOVED
site_properties	search_collapse_options	REMOVED
site_properties	disable_folder_section	SPECIAL
site_properties	disable_nonfolderish_sections	REMOVED
site_properties	typesUseViewActionInListings	plone.types_use_view_action_in_listings
site_properties	verify_login_name	plone.verify_login_name
site_properties	many_users	plone.many_users
site_properties	many_groups	plone.many_groups
site_properties	enable_livesearch	plone.enable_livesearch
site_properties	default_page_types	TBD
site_properties	use_folder_contents	REMOVED
site_properties	forbidden_contenttypes	TBD
site_properties	default_contenttype	REMOVED
site_properties	enable_sitemap	plone.enable_sitemap
site_properties	number_of_days_to_keep	REMOVED
site_properties	enable_inline_editing	REMOVED
site_properties	lock_on_ttw_edit	plone.lock_on_ttw_edit
site_properties	enable_link_integrity_checks	plone.enable_link_integrity_checks
site_properties	webstats_js	plone.webstats_js
site_properties	external_links_open_new_window	TBD
site_properties	icon_visibility	plone.icon_visibility
site_properties	mark_special_links	TBD
site_properties	redirect_links	TBD

Continued on next page

Table 5.1 – continued from previous page

site_properties	use_email_as_login	plone.use_email_as_login
site_properties	user_registration_fields	SPECIAL
site_properties	allow_external_login_sites	plone.allow_external_login_sites
site_properties	external_login_url	plone.external_login_url
site_properties	external_logout_url	plone.external_logout_url
site_properties	external_login_iframe	plone.external_login_iframe

disable_folder_sections

This property has been removed and the logic is different. You can influence the portal tab generation with the property *plone.generate_tabs*. This controls, if the tabs are generated from the content in the root folder. In addition, you can control if non folders will create entries or not with the property *plone.nonfolderish_tabs*. If you want to disable_folder_sections, you will want to set *plone.generate_tabs* to false.

Generic Setup

All settings for control panels are stored in the registry.xml Generic Setup file. This file can be exported through the Management Interface. Go to the Plone Site Setup, choose “Management Interface” from the “Advanced” section. Click on “portal_setup”. Go to the “export” tab. Choose the “Export the configuration registry schemata” check-box and click the “Export selected steps” button. The registry.xml file will contain entries like this:

```
<record name="plone.available_editors"
  interface="Products.CMFPlone.interfaces.controlpanel.IEditingSchema" field=
  ↪ "available_editors">
  <value>
    <element>TinyMCE</element>
    <element>None</element>
  </value>
</record>

<record name="plone.available_languages" interface="Products.CMFPlone.interfaces.
  ↪ controlpanel.ILanguageSchema" field="available_languages">
  <value>
    <element>en-us</element>
  </value>
</record>
```

Drop the settings you want to change into registry.xml in you Generic Setup profile folder. Re-install your add-on product and the settings will be available.

Python Code

All Generic Setup settings can be looked up with Python code.

First we lookup the registry utility:

```
>>> from zope.component import getUtility
>>> from plone.registry.interfaces import IRegistry
>>> registry = getUtility(IRegistry)
```

Now we use the schema ‘ISearchSchema’ to lookup for a RecordProxy object with all fields:

```
>>> from Products.CMFPlone.interfaces import ISearchSchema
>>> search_settings = registry.forInterface(ISearchSchema, prefix='plone')
```

Now we can get and set all fields of the schema above like:

```
>>> search_settings.enable_livesearch
True
```

If you want to change a setting, change the attribute:

```
>>> search_settings.enable_livesearch = False
```

Now the enable_livesearch should be disabled:

```
>>> search_settings.enable_livesearch
False
```

Editing Control Panel

Plone 5.x:

```
>>> from Products.CMFPlone.interfaces import IEditingSchema
>>> editing_settings = registry.forInterface(IEditingSchema, prefix='plone')

>>> editing_settings.default_editor
u'TinyMCE'

>>> editing_settings.ext_editor
False

>>> editing_settings.enable_link_integrity_checks
True

>>> editing_settings.lock_on_ttw_edit
True
```

Language Control Panel

All settings were managed with the tool *portal_languages* and with the GenericSetup file *portal_languages.xml*. Now these attributes are managed with Plone properties. As Plone 5 has full migration during an upgrade, please perform the upgrade and export the registry settings in GenericSetup to get the right settings. If you access attributes directly in your code, you must change your accessors. You know already how to get attributes from the *portal_languages* tool. The new attributes can be accessed via *plone.api* as described above.

old attribute	new attribute
root.portal_languages.supported_langs	plone.available_languages
site.portal_properties.site_properties.default_language or site.default_language	plone.default_language
root.portal_languages.use_combined_language_codes	plone.use_combined_language_codes
root.portal_languages.display_flags	plone.display_flags
portal_languages.use_path_negotiation	plone.use_path_negotiation
portal_languages.use_content_negotiation	plone.use_content_negotiation
portal_languages.use_cookie_negotiation	plone.use_cookie_negotiation
portal_languages.set_cookie_everywhere	plone.set_cookie_always
portal_languages.authenticated_users_only	plone.authenticated_users_only
portal_languages.use_request_negotiation	plone.use_request_negotiation
portal_languages.use_cctld_negotiation	plone.use_cctld_negotiation
portal_languages.use_subdomain_negotiation	plone.use_subdomain_negotiation
portal_languages.always_show_selector	plone.always_show_selector

Plone 5.x:

```
>>> from Products.CMFPlone.interfaces import ILanguageSchema
>>> language_settings = registry.forInterface(ILanguageSchema, prefix='plone')

>>> language_settings.available_languages
['en']
```

Mail Control Panel

All settings were managed with the tool *MailHost* and with the GenericSetup file `portal_languages.xml`. Now these attributes are managed with Plone properties. As Plone 5 has full migration during an upgrade, please perform the upgrade and export the registry settings in GenericSetup to get the right settings. If you access attributes directly in your code, you must change your accessors. You know already how to get attributes from the *portal_languages* tool. The new attributes can be accessed via `plone.api` as described above.

old attribute	new attribute
MailHost.smtp_host	plone.smtp_host
MailHost.smtp_port	plone.smtp_port
MailHost.smtp_user_id	plone.smtp_user_id
MailHost.smtp_pass	plone.smtp_pass
MailHost.email_from_address	plone.email_from_address
MailHost.email_from_name	plone.email_from_name

Maintenance Control Panel

Plone 5.x:

```
>>> from Products.CMFPlone.interfaces import IMaintenanceSchema
>>> maintenance_settings = registry.forInterface(IMaintenanceSchema, prefix='plone')

>>> maintenance_settings.days
7
```

Navigation Control Panel

Plone 5.x:

```
>>> from Products.CMFPlone.interfaces import INavigationSchema
>>> navigation_settings = registry.forInterface(INavigationSchema, prefix='plone')

>>> navigation_settings.generate_tabs
True

>>> navigation_settings.nonfolderish_tabs
True

>>> navigation_settings.displayed_types
('Image', 'File', 'Link', 'News Item', 'Folder', 'Document', 'Event')

>>> navigation_settings.filter_on_workflow
False

>>> navigation_settings.workflow_states_to_show
()

>>> navigation_settings.show_excluded_items
True
```

Search Control Panel

Plone 5.x:

```
>>> from Products.CMFPlone.interfaces import ISearchSchema
>>> search_settings = registry.forInterface(ISearchSchema, prefix='plone')

>>> search_settings.enable_livesearch
False

>>> search_settings.types_not_searched
(...)
```

Site Control Panel

Plone 4.x:

```
>>> portal = getSite()
>>> portal_properties = getToolByName(portal, "portal_properties")
>>> site_properties = portal_properties.site_properties

>>> portal.site_title = settings.site_title
>>> portal.site_description = settings.site_description
>>> site_properties.enable_sitemap = settings.enable_sitemap
>>> site_properties.exposeDCMetaTags = settings.exposeDCMetaTags
>>> site_properties.webstats_js = settings.webstats_js

>>> settings.enable_sitemap -> plone.app.layout
```

Plone 5.x:

```
>>> from Products.CMFPlone.interfaces import ISiteSchema
>>> site_settings = registry.forInterface(ISiteSchema, prefix='plone')

>>> site_settings.site_title
u'Plone site'

>>> site_settings.exposeDCMetaTags
False

>>> site_settings.enable_sitemap
False

>>> site_settings.webstats_js
u''
```

Overview Control Panel

Plone 5.x:

```
>>> from Products.CMFPlone.interfaces.controlpanel import IDateAndTimeSchema
>>> tz_settings = registry.forInterface(IDateAndTimeSchema, prefix='plone')

>>> tz_settings.portal_timezone = 'UTC'
```

Markup Control Panel

Plone 5.x:

```
>>> from Products.CMFPlone.interfaces import IMarkupSchema
>>> markup_settings = registry.forInterface(IMarkupSchema, prefix='plone')

>>> markup_settings.default_type
u'text/html'

>>> markup_settings.allowed_types
('text/html', 'text/x-web-textile')
```

User and Groups Control Panel

Plone 5.x:

```
>>> from Products.CMFPlone.interfaces import IUserGroupsSettingsSchema
>>> usergroups_settings = registry.forInterface(IUserGroupsSettingsSchema, prefix=
↳ 'plone')

>>> usergroups_settings.many_groups
False

>>> usergroups_settings.many_users
False
```


portal_languages is now a utility

Part of the work on PLIP 13091 (plone.app.multilingual) required to move `portal_languages` to a utility.

So code that used to look like this:

```
# OLD 4.x approach
portal.portal_languages.getDefaultLanguage()
```

Now it should look like this:

```
# NEW in 5.0
language_tool = api.portal.get_tool('portal_languages')
language_tool.getDefaultLanguage()
```

Tests changes

In Plone 4.x a date or date time widget used to be rendered as a set of input fields:

```
# OLD 4.x approach
browser_manager.getControl(name='form.widgets.IPublication.effective-year').value =
↪ '2015'
browser_manager.getControl(name='form.widgets.IPublication.effective-month').value = [
↪ '10']
browser_manager.getControl(name='form.widgets.IPublication.effective-day').value = '11'
↪ '
browser_manager.getControl(name='form.widgets.IPublication.effective-hour').value =
↪ '15'
browser_manager.getControl(name='form.widgets.IPublication.effective-min').value = '14'
↪ '
```

Now the same input field will be rendered as a single string input:

```
# NEW in 5.0
browser_manager.getControl(name='form.widgets.IPublication.effective').value = '2015-
↪ 10-11 15:14'
```

Deprecation of `portal_properties.xml`

`portal_properties.xml` Generic Setup import step is now deprecated and has been moved to `plone.registry`.

parentMetaTypesNotToQuery

```
# OLD 4.x approach
<object name="portal_properties">
  <object name="navtree_properties">
    <property name="parentMetaTypesNotToQuery" purge="false">
      <element value="my.hidden.content.type" />
    </property>
  </object>
</object>
```

Now in `registry.xml` should look like:

```
# NEW in 5.0
<?xml version="1.0"?>
<registry>
  <record
    name="plone.parent_types_not_to_query"
    interface="Products.CMFPlone.interfaces.controlpanel.INavigationSchema"
    field="parent_types_not_to_query">
    <value>
      <element value="my.hidden.content.type" />
    </value>
  </record>
</registry>
```

metaTypesNotToList

```
# OLD 4.x approach
<?xml version="1.0"?>
<object name="portal_properties">
  <object name="navtree_properties">
    <property name="metaTypesNotToList" purge="false">
      <element value="my.hidden.content.type" />
    </property>
  </object>
</object>
```

nothing should be done in Plone 5.

The new setting is on `Products.CMFPlone.interfaces.controlpanel.INavigationSchema.displayed_types` and it works the other way around.

Instead of blacklisting content types it whitelists them, if you don't want your content type to show there's nothing to do.

typesLinkToFolderContentsInFC

```
# OLD 4.x approach
<?xml version="1.0"?>
<object name="portal_properties">
  <object name="site_properties">
    <property name="typesLinkToFolderContentsInFC" purge="false">
      <element value="my.fancy.content.type" />
    </property>
  </object>
</object>
```

Now in `registry.xml` should look like:

```
# NEW in Plone 5
<record
  name="plone.types_use_view_action_in_listings"
  interface="Products.CMFPlone.interfaces.controlpanel.ITypesSchema"
  field="types_use_view_action_in_listings">
  <value>
    <element>my.fancy.content.type</element>
  </value>
</record>
```

types_not_searched

```
# OLD 4.x approach
<?xml version="1.0"?>
<object name="portal_properties">
  <object name="site_properties">
    <property name="types_not_searched" purge="false">
      <element value="my.fancy.content.type" />
    </property>
  </object>
</object>
```

Now in registry.xml should look like:

```
# NEW in Plone 5
<?xml version="1.0"?>
<registry>
  <record
    name="plone.types_not_searched"
    interface="Products.CMFPlone.interfaces.controlpanel.ISearchSchema"
    field="types_not_searched">
    <value>
      <element>my.fancy.content.type</element>
    </value>
  </record>
</registry>
```

Troubleshooting

Basic troubleshooting

Description

Here is some info for basic Plone troubleshooting, especially with add-on modules

Start Plone as foreground / debug mode

Plone runs on the top of Zope application service. Zope is a Python process and will appear as “python” in your task manager process list.

Zope will report any problems with code when it is launched in foreground mode (attached to a terminal).

- Basic command-line knowledge is needed in order to proceed

First stop Zope if it is running as a background process

- On Windows: use Plone Control Panel or Windows Control Panel Services section to shutdown Plone first
- On Linux: use /etc/init.d/plone stop or related command to shutdown Plone

Use the command

```
bin/instance fg
```

or Windows command-line command (note that Plone location may depend on where you installed it)

```
cd "C:\Program files\Plone"  
bin\instance.exe fg
```

to start Plone.

All errors will be printed into the terminal. The error is printed as Python *traceback*. It is important to copy-paste all lines of this traceback, not just the last line.

If there is no start up error you will see the line:

```
INFO Zope Ready to handle requests
```

as the last line of the startup sequence.

No such file or directory: 'zope.conf'

Example:

```
sudo /Applications/Plone/zinstance/bin/plonectl start  
instance: Error: error opening file /Applications/Plone/zinstance/parts/instance/etc/  
↪zope.conf: [Errno 2] No such file or directory: '/Applications/Plone/zinstance/  
↪parts/instance/etc/zope.conf'
```

This means that running `bin/buildout` script did not complete successfully. Re-run buildout and see what's wrong.

Dropping into pdb

If you need to inspect start-up errors in Python's *debugger*.

Activate Python configuration associated with your `bin/instance` script:

```
source ~/code/collective.buildout.python/python-2.6/bin/activate
```

Start Plone pdb enabled:

```
python -m pdb bin/instance fg
```

Check if Plone is up and responds to requests

Enter to the computer running Plone (SSH in on UNIX).

Use `telnet` command to connect Plone port and see if you get valid HTTP response from Plone

```
telnet localhost 8080
```

Then do a human HTTP user agent simulation by typing:

```
GET / HTTP/1.0<enter><enter>
```

Plone response looks like this:

```

Trying 127.0.0.1...
Connected to localhost.localdomain.
Escape character is '^]'.
GET / HTTP/1.0

HTTP/1.0 200 OK
Server: Zope/(2.13.10, python 2.6.6, linux2) ZServer/1.1
Date: Wed, 01 Feb 2012 09:59:40 GMT
Content-Length: 1614
Content-Type: text/html; charset=utf-8
Connection: close

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">

<head>
<base href="http://xxx.fi:9980/" />

```

If you get the answer from Plone (based on HTTP response headers) then Plone is running and you have problem elsewhere in your firewall/server/front-end web server configuration.

Consult your operating system manual for fixing your problem.

Cleaning up bad add-on uninstalls

Many low quality Plone add-ons do not uninstall cleanly.

You need to remove persistent objects from the site database *after* add-on uninstall while *code is still in buildout*.

Otherwise your Plone site may not

- Pack properly
- Start properly
- Migrate to new version

For more information see *Manually Removing Local Persistent Utilities*

Not able to log in

It might happen that you start your instance with an empty database and you are not able to log in even if you are absolutely sure about your password. If you work on localhost throw away the localhost related cookies in your browser and restart.

If you have lost the Zope Admin Password you can create an emergency user:

- <http://quintagroup.com/services/support/tutorials/zope-access>

More info

- *common exceptions which you might encounter when starting Zope*
- Plone community support guidelines for asking help

Exceptions and common tracebacks

Description

Common Python exception traceback patterns you may encounter when working with Plone and possible solutions for them.

Please see [this tutorial](#) for extracting Python tracebacks from your Plone logs.

Add-on installer error: This object was originally created by a product that is no longer installed

Traceback:

```
2009-10-18 13:11:20 ERROR Zope.SiteErrorLog 1255860680.760.514176531634 http://
↳localhost:8080/twinapex/portal_quickinstaller/installProducts
Traceback (innermost last):
  Module ZPublisher.Publish, line 125, in publish
  Module Zope2.App.startup, line 238, in commit
  Module transaction._manager, line 93, in commit
  Module transaction._transaction, line 325, in commit
  Module transaction._transaction, line 424, in _commitResources
  Module ZODB.Connection, line 541, in commit
  Module ZODB.Connection, line 586, in _commit
  Module ZODB.Connection, line 620, in _store_objects
  Module ZODB.serialize, line 407, in serialize
  Module OFS.Uninstalled, line 40, in __getstate__
SystemError: This object was originally created by a product that
              is no longer installed.  It cannot be updated.
              (<Salt at broken>)
```

Reason: Data.fs contains objects for which the code is not present. You have probably moved Data.fs or edited buildout.cfg.

Solution: Check that eggs and zcml contain all necessary products in buildout.cfg.

See also:

- <http://article.gmane.org/gmane.comp.web.zope.plone.setup/3232>

Add-on installer error: too many values to unpack

Traceback:

```
Module ZPublisher.Publish, line 119, in publish
Module ZPublisher.mapply, line 88, in mapply
Module ZPublisher.Publish, line 42, in call_object
Module Products.CMFQuickInstallerTool.QuickInstallerTool, line 589, in installProducts
Module Products.CMFQuickInstallerTool.QuickInstallerTool, line 475, in installProduct
- __traceback_info__: ('gomobile.mobile',)
Module Products.CMFQuickInstallerTool.QuickInstallerTool, line 396, in snapshotPortal
Module five.localsitemanager.registry, line 194, in registeredUtilities
Module zope.component.registry, line 127, in registeredUtilities
ValueError: too many values to unpack
```

Condition: When trying to install a plugin

Reason: You have run Data.fs with zope.component 3.5.1, but later downgraded / moved Data.fs.

Solution: Pin zope.component to 3.5.1.

Archetypes: TypeError: getattr(): attribute name must be string

Traceback:

```
'user': <PropertiedUser 'admin'>}
Module Products.PageTemplates.ZRPythonExpr, line 48, in __call__
- __traceback_info__: otherwidget.Description(here, target_language=target_language)
Module PythonExpr, line 1, in <expression>
Module Products.Archetypes.generator.widget, line 100, in Description
TypeError: getattr(): attribute name must be string
```

Reason: You might have used something else besides string or translation string to define Archetypes widget name or description.

AttributeError in setRoles due to workflow state transition

Traceback:

```
Traceback (innermost last):
Module ZPublisher.Publish, line 115, in publish
Module ZPublisher.mapply, line 88, in mapply
Module ZPublisher.Publish, line 41, in call_object
Module Products.CMFPlone.FactoryTool, line 361, in __call__
Module Products.CMFPlone.FactoryTool, line 147, in __getitem__
Module Products.CMFPlone.PloneFolder, line 406, in invokeFactory
Module Products.CMFCore.TypesTool, line 934, in constructContent
Module Products.CMFCore.TypesTool, line 345, in constructInstance
Module Products.CMFCore.TypesTool, line 357, in _finishConstruction
Module Products.CMFCore.CMFCatalogAware, line 145, in notifyWorkflowCreated
Module Products.CMFCore.WorkflowTool, line 355, in notifyCreated
Module Products.DCWorkflow.DCWorkflow, line 392, in notifyCreated
Module Products.DCWorkflow.DCWorkflow, line 476, in _changeStateOf
Module Products.DCWorkflow.DCWorkflow, line 571, in _executeTransition
Module Products.DCWorkflow.DCWorkflow, line 435, in updateRoleMappingsFor
Module Products.DCWorkflow.utils, line 60, in modifyRolesForPermission
Module AccessControl.Permission, line 93, in setRoles
AttributeError: appname
```

Possible reasons:

1. You are using AnnotationStorage but you forgot to declare atapi.ATFieldProperty in your class body
2. You are inhering schema in Archetypes, but you do not inherit the class itself

AttributeError: 'FileSystemResourceDirectory' object has no attribute 'absolute_url'

Traceback:

```
2013-09-02 12:26:55 ERROR plone.transformchain Unexpected error whilst trying to_
↳ apply transform chain
Traceback (most recent call last):
  File "/home/pab/.buildout/eggs/plone.transformchain-1.0.3-py2.7.egg/plone/
↳ transformchain/transformer.py", line 48, in __call__
```

```
newResult = handler.transformIterable(result, encoding)
File "/home/pab/.buildout/eggs/plone.app.theming-1.1.1-py2.7.egg/plone/app/theming/
↳transform.py", line 179, in transformIterable
    params = prepareThemeParameters(findContext(self.request), self.request,
↳parameterExpressions, cache)
File "/home/pab/.buildout/eggs/plone.app.theming-1.1.1-py2.7.egg/plone/app/theming/
↳utils.py", line 630, in prepareThemeParameters
    params[name] = quote_param(expression(expressionContext))
File "/home/pab/.buildout/eggs/Zope2-2.13.20-py2.7.egg/Products/PageTemplates/
↳ZRPythonExpr.py", line 48, in __call__
    return eval(self._code, vars, {})
File "PythonExpr", line 1, in <expression>
File "/home/pab/.buildout/eggs/plone.memoize-1.1.1-py2.7.egg/plone/memoize/view.py",
↳line 47, in memogetter
    value = cache[key] = func(*args, **kwargs)
File "/home/pab/.buildout/eggs/plone.app.layout-2.3.5-py2.7.egg/plone/app/layout/
↳globals/context.py", line 47, in current_base_url
    self.context.absolute_url()))
AttributeError: 'FileSystemResourceDirectory' object has no attribute 'absolute_url'
```

Reason: There is a not accessible filesystem resource declared in your diazo theme's html.

Solution: Check that all js and css files are available.

AttributeError: 'RelationList' object has no attribute 'source'

Traceback:

```
2014-03-21 17:19:09 ERROR Zope.SiteErrorLog 1395433149.260.697467198696 http://
↳localhost:8080/Plone/++add++MyType
Traceback (innermost last):
  Module ZPublisher.Publish, line 138, in publish
  Module ZPublisher.mapply, line 77, in mapply
  Module ZPublisher.Publish, line 48, in call_object
  Module plone.z3cform.layout, line 66, in __call__
  Module plone.z3cform.layout, line 50, in update
  Module plone.dexterity.browser.add, line 112, in update
  Module plone.z3cform.fieldsets.extensible, line 59, in update
  Module plone.z3cform.patch, line 30, in GroupForm_update
  Module z3c.form.group, line 128, in update
  Module z3c.form.form, line 134, in updateWidgets
  Module z3c.form.field, line 277, in update
  Module z3c.formwidget.query.widget, line 108, in update
  Module z3c.formwidget.query.widget, line 95, in bound_source
  Module z3c.formwidget.query.widget, line 90, in source
AttributeError: 'RelationList' object has no attribute 'source'
```

Reason: You're trying to use a relation field on your Dexterity-based content type but `plone.app.relationfield` is not installed.

Solution: Follow the instructions on the Dexterity documentation as `relation` support is no longer included by default.

AttributeError: 'module' object has no attribute 'HTTPSConnection'

Python has not been compiled with HTTPS support.

Try installing your Python, for example, using minitage.

See *Python basics*.

AttributeError: 'str' object has no attribute 'other' (Mixed zope.viewpagetemplate and Five.viewpagetemplate)

Traceback:

```
Module zope.tales.ales, line 696, in evaluate
- URL: /home/moo/sits/src/plone.z3cform/plone/z3cform/crud/crud-master.pt
- Line 17, Column 2
- Expression: <PathExpr standard:u'form/render'>
- Names:
  {'args': (),
   'context': <SitsPatient at /folder_sits/sitsngta/intranet/sitsdatabase/
↪sitscountry_TE/sitshospital_TES/sitspatient.TETES2009062217>,
   'default': <object object at 0xb7d76538>,
   'loop': {},
   'nothing': None,
   'options': {},
   'repeat': {},
   'request': <HTTPRequest, URL=http://localhost:9000/folder_sits/sitsngta/intranet/
↪sitsdatabase/sitscountry_TE/sitshospital_TES/sitspatient.TETES2009062217/@@ar>,
   'template': <zope.app.pagetemplate.viewpagetemplatefile.ViewPageTemplateFile_
↪object at 0xc6e552c>,
   'usage': <zope.pagetemplate.pagetemplate.TemplateUsage object at 0xf7fb78c>,
   'view': <Products.SitsPatient.browser.ar.ARCrudForm object at 0xf928ccc>,
   'views': <zope.app.pagetemplate.viewpagetemplatefile.ViewMapper object at_
↪0xf7b4a0c>}
Module Products.PTPProfiler.ProfilerPatch, line 32, in __patched_call__
Module zope.tales.expressions, line 217, in __call__
Module zope.tales.expressions, line 211, in _eval
Module z3c.form.form, line 143, in render
Module Shared.DC.Scripts.Bindings, line 313, in __call__
Module Shared.DC.Scripts.Bindings, line 348, in _bindAndExec
Module Shared.DC.Scripts.Bindings, line 1, in ?
Module Shared.DC.Scripts.Bindings, line 293, in _getTraverseSubpath
AttributeError: 'str' object has no attribute 'other'
```

Five ViewPageTemplate class file is slightly different than Zope 3's normal ViewPageTemplate file. In this case Five ViewPageTemplate was used, when Zope 3's normal ViewPageTemplate was expected.

Another reason is that acquisition chain is not properly set-up in your custom views.

Difference:

```
from Products.Five.browser.pagetemplatefile import ViewPageTemplateFile
```

vs.:

```
from zope.pagetemplate.pagetemplatefile import PageTemplateFile
```

AttributeError: 'wrapper_descriptor' object has no attribute 'im_func'

Traceback:

```
File "/home/moo/code/gomobile/parts/zope2/lib/python/DocumentTemplate/DT_Util.py",  
↳line 19, in <module>  
    from html_quote import html_quote, ustr # for import by other modules, dont remove!  
File "/home/moo/code/gomobile/parts/zope2/lib/python/DocumentTemplate/html_quote.py",  
↳line 4, in <module>  
    from ustr import ustr  
File "/home/moo/code/gomobile/parts/zope2/lib/python/DocumentTemplate/ustr.py", line  
↳18, in <module>  
    nasty_exception_str = Exception.__str__.im_func  
AttributeError: 'wrapper_descriptor' object has no attribute 'im_func'
```

Condition: This exception happens when starting Plone

Reason: You are trying to use Python 2.6 with Plone 3

Solution: With Plone 3 you need to use Python 2.4.

AttributeError: REQUEST in getObject

Traceback:

```
import ZPublisher, Zope  
Traceback (most recent call last):  
  File "<string>", line 1, in ?  
  File "src/collective.mountpoint/collective/mountpoint/bin/update.py", line 31, in ?  
    sys.exit(main(app))  
  File "/srv/plone/saariselka/src/collective.mountpoint/collective/mountpoint/  
↳updateclient.py", line 243, in main  
    exit_code = updater.updateAll()  
  File "/srv/plone/saariselka/src/collective.mountpoint/collective/mountpoint/  
↳updateclient.py", line 151, in updateAll  
    mountpoints = list(self.getMountPoints())  
  File "/srv/plone/saariselka/src/collective.mountpoint/collective/mountpoint/  
↳updateclient.py", line 49, in getMountPoints  
    return [ brain.getObject() for brain in brains ]  
  File "/srv/plone/saariselka/parts/zope2/lib/python/Products/ZCatalog/CatalogBrains.  
↳py", line 86, in getObject  
    target = parent.restrictedTraverse(path[-1])  
  File "/srv/plone/saariselka/parts/zope2/lib/python/OFS/Traversable.py", line 301,  
↳in restrictedTraverse  
    return self.unrestrictedTraverse(path, default, restricted=True)  
  File "/srv/plone/saariselka/parts/zope2/lib/python/OFS/Traversable.py", line 259,  
↳in unrestrictedTraverse  
    next = queryMultiAdapter((obj, self.REQUEST),  
AttributeError: REQUEST
```

Reason: You are using command line script. getObject() fails for a catalog brain, because the actual object is gone. However, unrestrictedTraverse() does not handle this case gracefully.

AttributeError: Schema

Traceback:

```
Module zope.tales.traits, line 696, in evaluate  
- URL: file:/fast/xxm2011/eggs/Products.Archetypes-1.7.10-py2.6.egg/Products/  
↳Archetypes/skins/archetypes/base_view.pt
```

```

- Line 50, Column 4
- Expression: <PythonExpr context.Schema().viewableFields(here)>
- Names:
  {'container': <CourseInfo at /xxx/courses/professional-courses/business-
  ↪management-courses/postgraduate-diploma-in-business-and-management-consultancy>,
   'context': <CourseInfo at /xxx/courses/professional-courses/business-management-
  ↪courses/postgraduate-diploma-in-business-and-management-consultancy>,
   'default': <object object at 0x1002edb70>,
   'here': <CourseInfo at /xxx/courses/professional-courses/business-management-
  ↪courses/postgraduate-diploma-in-business-and-management-consultancy>,
   'loop': {},
   'nothing': None,
   'options': {'args': ()},
   'repeat': <Products.PageTemplates.Expressions.SafeMapping object at 0x10b70a208>,
   'request': <HTTPRequest, URL=http://localhost:8090/xxx/courses/professional-
  ↪courses/business-management-courses/postgraduate-diploma-in-business-and-management-
  ↪consultancy/base_view>,
   'root': <Application at >,
   'template': <FSPageTemplate at /xxx/courses/professional-courses/business-
  ↪management-courses/postgraduate-diploma-in-business-and-management-consultancy/base_
  ↪view>,
   'traverse_subpath': [],
   'user': <PropertiedUser 'admin'>}
Module Products.PageTemplates.ZRPythonExpr, line 48, in __call__
- __traceback_info__: context.Schema().viewableFields(here)
Module PythonExpr, line 1, in <expression>
Module AccessControl.ImplPython, line 675, in guarded_getattr

```

Condition: This error may come when you try to view your custom content type

Reason: It is picking up Archetypes default view template for your Dexterity content type.

Try if you can access your view by directly calling it to by its name. E.g.:

```
http://yoursite.com/folder/content/@@view
```

If it's working then it is wrong data in *portal_types*.

Your content item might also be corrupted. It is trying to use dynamic view selector even if it's not supported. Try re-creating the particular content item.

AttributeError: getPhysicalPath()

Traceback:

```

Module zope.tal.talinterpreter, line 408, in do_startTag
Module zope.tal.talinterpreter, line 485, in attrAction_tal
Module Products.PageTemplates.Expressions, line 230, in evaluateText
Module zope.tales.tales, line 696, in evaluate
- URL: edit_header
- Line 25, Column 14
- Expression: <PythonExpr (view.getHeaderDefiner().absolute_url())>
- Names:
  {'container': <Frontpage at /yourinstance/matkailijalle/yourinstance-1>,
   'context': <Frontpage at /yourinstance/matkailijalle/yourinstance-1>,
   'default': <object object at 0x7fabf9ceclf0>,
   'here': <Frontpage at /yourinstance/matkailijalle/yourinstance-1>,
   'loop': {},

```

```
'nothing': None,
'options': {'args': ()},
'repeat': <Products.PageTemplates.Expressions.SafeMapping object at 0xe617d88>,
'request': <HTTPRequest, URL=http://localhost:9444/yourinstance/matkailijalle/
↳yourinstance-1/@@edit_header>,
'root': <Application at >,
'template': <ImplicitAcquirerWrapper object at 0xe6105d0>,
'traverse_subpath': [],
'user': <PropertyUser 'admin'>,
'view': <Products.Five.metaclass.EditHeaderBehaviorView object at 0xe51ed10>,
'views': <zope.app.pagetemplate.viewpagetemplatefile.ViewMapper object at
↳0xe610c10>}
Module zope.tales.pythonexpr, line 59, in __call__
- __traceback_info__: (view.getHeaderDefiner().absolute_url())
Module <string>, line 0, in ?
Module OFS.Traversable, line 64, in absolute_url
Module OFS.Traversable, line 117, in getPhysicalPath
AttributeError: getPhysicalPath
```

Another possible error is:

```
AttributeError: absolute_url
```

This usually means that you should have used `context.aq_inner` when you have used `context.absolute_url()` tries to get the path to the object, but object parent is set to view (`context.aq_parent`) instead of real container object (`context.aq_inner.aq_parent`).

Warning: When setting a member attribute in `BrowserView`, the acquisition parent of objects changes to `BrowserView` instance. All member attributes receive `ImplicitAcquisitionWrapper` automatically.

Demonstration

We try to set `BrowserView` member attribute `defining_context` to be some context object.:

```
(Pdb) self.defining_context = context
(Pdb) context.aq_parent
<PloneSite at /plone>
(Pdb) self.defining_context.aq_parent
<Products.Five.metaclass.HeaderAnimationHelper object at 0xad5750>
(Pdb) self.defining_context.aq_inner.aq_parent
<Products.Five.metaclass.HeaderAnimationHelper object at 0xad5750>
(Pdb) self.defining_context.aq_parent.aq_parent
<ATDocument at /plone/doc>
(Pdb) self.defining_context.aq_parent.aq_parent.aq_inner
<ATDocument at /plone/doc>
(Pdb) self.defining_context.aq_parent.aq_parent.aq_parent
<PloneSite at /plone>
```

To get the real object (as it was before `set` was called) you can create a helper getter:

```
def getDefiningContext(self):
    """
    Un-fuse automatically injected view from the acquisition chain

    @return: Real defining context object without bad acquisition
    """
```

```

if self.defining_context is not None:
    return self.defining_context.aq_parent.aq_inner.aq_parent
return None

```

AttributeError: type object 'IRAMCache' has no attribute '__iro__'

Traceback:

```

Module zope.component._api, line 130, in subscribers
Module zope.component.registry, line 290, in subscribers
Module zope.interface.adapter, line 535, in subscribers
Module zope.app.component.site, line 375, in threadSiteSubscriber
Module zope.app.component.hooks, line 61, in setSite
Module Products.CMFCore.PortalObject, line 75, in getSiteManager
Module ZODB.Connection, line 811, in setstate
Module ZODB.Connection, line 870, in _setstate
Module ZODB.serialize, line 605, in setGhostState
Module zope.component.persistentregistry, line 42, in __setstate__
Module zope.interface.adapter, line 80, in _createLookup
Module zope.interface.adapter, line 389, in __init__
Module zope.interface.adapter, line 426, in init_extensors
Module zope.interface.adapter, line 430, in add_extendor
AttributeError: type object 'IRAMCache' has no attribute '__iro__'

```

Condition: This error can happen when trying to open any page

Reason: You have probably imported a Data.fs using newer Plone/Zope version to old Plone, or package pindowns are incorrect. If you are copying a site try re-checking that source and target buildouts and package versions match.

AttributeError: set_stripped_tags

Traceback:

```

...
Module ZPublisher.Publish, line 60, in publish
Module ZPublisher.mapply, line 77, in mapply
Module ZPublisher.Publish, line 46, in call_object
Module zope.formlib.form, line 795, in __call__
Module five.formlib.formbase, line 50, in update
Module zope.formlib.form, line 776, in update
Module zope.formlib.form, line 620, in success
Module plone.app.controlpanel.form, line 38, in handle_edit_action
Module zope.formlib.form, line 543, in applyChanges
Module zope.formlib.form, line 538, in applyData
Module zope.schema._bootstrapfields, line 227, in set
Module plone.app.controlpanel.filter, line 173, in set_
AttributeError: set_stripped_tags

```

Condition: This error may happen on saving changed settings in the HTML-Filtering controlpanel.

possible cause:

- You have migrated your Plone site from 3.3.5 to Plone 4.x
- For some reason kupu library tool may not be removed in the upgrade step that removed kupu.

Solution: Go to the Management Interface and delete the kupu library tool manually.

AttributeError: set_stripped_combinations

Traceback:

```
...
Module ZPublisher.Publish, line 126, in publish
Module ZPublisher.mapply, line 77, in mapply
Module ZPublisher.Publish, line 46, in call_object
Module zope.formlib.form, line 795, in __call__
Module five.formlib.formbase, line 50, in update
Module zope.formlib.form, line 776, in update
Module zope.formlib.form, line 620, in success
Module plone.app.controlpanel.form, line 38, in handle_edit_action
Module zope.formlib.form, line 543, in applyChanges
Module zope.formlib.form, line 538, in applyData
Module zope.schema._bootstrapfields, line 227, in set
Module plone.app.controlpanel.filter, line 254, in set
AttributeError: set_stripped_combinations
```

Condition: This error may happen on saving changed settings in the HTML-Filtering controlpanel.

possible cause:

- You have migrated your Plone site from 3.3.5 to Plone 4.x
- For some reason kupu library tool may not be removed in the upgrade step that removed kupu.

Solution: Go to the Management Interface and delete the kupu library tool manually.

BadRequest: The id “xxx” is invalid - it is already in use.

Traceback:

```
...
Module Products.CMFFormController.Script, line 145, in __call__
Module Products.CMFCore.FSPythonScript, line 140, in __call__
Module Shared.DC.Scripts.Bindings, line 313, in __call__
Module Shared.DC.Scripts.Bindings, line 350, in _bindAndExec
Module Products.CMFCore.FSPythonScript, line 196, in _exec
Module None, line 1, in content_edit
<FSControllerPythonScript at /xxx/content_edit used for /xxx/sisalto/lomapalvelut/
↳portal_factory/HolidayService/aktiviteetit>
Line 1
Module Products.CMFCore.FSPythonScript, line 140, in __call__
Module Shared.DC.Scripts.Bindings, line 313, in __call__
Module Shared.DC.Scripts.Bindings, line 350, in _bindAndExec
Module Products.CMFCore.FSPythonScript, line 196, in _exec
Module None, line 9, in content_edit_impl
<FSPythonScript at /xxx/content_edit_impl used for /xxx/sisalto/lomapalvelut/portal_
↳factory/HolidayService/aktiviteetit>
Line 9
Module Products.CMFPlone.FactoryTool, line 264, in doCreate
Module Products.ATContentTypes.lib.constrainttypes, line 281, in invokeFactory
Module Products.CMFCore.PortalFolder, line 315, in invokeFactory
Module Products.CMFCore.TypesTool, line 716, in constructContent
Module Products.CMFCore.TypesTool, line 276, in constructInstance
Module Products.CMFCore.TypesTool, line 450, in _constructInstance
Module xxx.app.content.holidayservice, line 7, in addHolidayService
Module OFS.ObjectManager, line 315, in _setObject
```

```
Module Products.CMFCore.PortalFolder, line 333, in _checkId
Module OFS.ObjectManager, line 102, in checkValidId
BadRequest: The id "holidayservice.2010-03-18.4474765045" is invalid - it is already_
↳ in use.
```

Try portal_catalog rebuild as a fix.

ComponentLookupError: cmf.ManagePortal

Traceback:

```
zope.configuration.config.ConfigurationExecutionError: <class 'zope.component.
↳ interfaces.ComponentLookupError'>: (<InterfaceClass zope.security.interfaces.
↳ IPermission>, u'cmf.ManagePortal')
in:
File "/fast/x/src/collective.portletcollection/collective/portletcollection/
↳ portlets/configure.zcml", line 11.2-20.8
```

Condition: This error may happen when starting Plone

This is a sign of changed loading order, starting from Plone 4.1. You need to explicitly include *CMF-Core/permissions.zcml* in your *configuration.zcml*.

Example:

```
<include package="Products.CMFCore" file="permissions.zcml" />
```

Content status history won't render - traceback is content path reversed

Traceback:

```
Module zope.tales.ales, line 696, in evaluate
- URL: file:/home/antti/workspace/plone/hotellilevitunturi/eggs/Plone-3.3.5-py2.4.
↳ egg/Products/CMFPlone/skins/plone_forms/content_status_history.cpt
- Line 201, Column 14
- Expression: <PythonExpr wtool.getTransitionsFor(target, here)>
- Names:
  {'container': <PloneSite at /hotellilevitunturi>,
   'context': <MainFolder at /hotellilevitunturi/fi/ravintolamaailma>,
   'default': <object object at 0xb75d2540>,
   'here': <MainFolder at /hotellilevitunturi/fi/ravintolamaailma>,
   'loop': {},
   'nothing': None,
   'options': {'args': (),
               'state': <Products.CMFFormController.ControllerState.ControllerState_
↳ object at 0x1055614c>},
   'repeat': <Products.PageTemplates.Expressions.SafeMapping object at 0x10556f6c>,
   'request': <HTTPRequest, URL=http://localhost:9888/hotellilevitunturi/fi/
↳ ravintolamaailma/content_status_history>,
   'root': <Application at >,
   'template': <FSControllerPageTemplate at /hotellilevitunturi/content_status_
↳ history used for /hotellilevitunturi/fi/ravintolamaailma>,
   'traverse_subpath': [],
   'user': <PropertiedUser 'admin'>}}
Module Products.PageTemplates.ZRPythonExpr, line 49, in __call__
- __traceback_info__: wtool.getTransitionsFor(target, here)
```

```
Module PythonExpr, line 1, in <expression>
Module Products.CMFPlone.WorkflowTool, line 88, in getTransitionsFor
Module Products.CMFPlone.WorkflowTool, line 37, in flattenTransitions
Module Products.CMFPlone.WorkflowTool, line 69, in flattenTransitionsForPaths
Module OFS.Traversable, line 301, in restrictedTraverse
Module OFS.Traversable, line 284, in unrestrictedTraverse
- __traceback_info__: ([u's', u'a', u'n', u'u', u'o', u'l', u '/', u'a', u'm', u'l', u
↪ 'i', u'a', u'a', u'm', u'a', u'l', u'o', u't', u'n', u'i', u'v', u'a', u'r', u '/', u
↪ 'i', u'f', u '/', u'i', u'r', u'u', u't', u'n', u'u', u't', u'i', u'v', u'e', u'l', u
↪ 'i', u'l', u'l', u'e', u't', u'o', u'h'], u '/')
KeyError: u '/'
```

ContentProviderLookupError: plone.htmlhead

Traceback:

```
Module zope.tales.ales, line 696, in evaluate
- URL: file:/home/moo/isleofback/eggs/Plone-3.3.5-py2.4.egg/Products/CMFPlone/skins/
↪ plone_templates/main_template.pt
- Line 39, Column 4
- Expression: <StringExpr u'plone.htmlhead'>
- Names:
  {'container': <PloneSite at /isleofback>,
   'context': <PloneSite at /isleofback>,
   'default': <object object at 0xb75f2528>,
   'here': <PloneSite at /isleofback>,
   'loop': {},
   'nothing': None,
   'options': {'args': (<isleofback.app.browser.company.CompanyCreationForm object _
↪ at 0xea5e80c>,)},
   'repeat': <Products.PageTemplates.Expressions.SafeMapping object at 0xea62dcc>,
   'request': <HTTPRequest, URL=http://localhost:9666/isleofback/@@create_company>,
   'root': <Application at >,
   'template': <ImplicitAcquirerWrapper object at 0xea62bcc>,
   'traverse_subpath': [],
   'user': <PropertiedUser 'admin'>,
   'view': <UnauthorizedBinding: context>,
   'views': <zope.app.pagetemplate.viewpagetemplatefile.ViewMapper object at _
↪ 0xea62d2c>}
Module Products.Five.browser.providerexpression, line 25, in __call__
ContentProviderLookupError: plone.htmlhead
```

This is not a bug in Zope. It is caused by trying to render a Plone page frame in an context which has not acquisition chain properly set up. Plone `main_template.pt` tries to look up viewlet managers by acquisition traversing to parent objects. `plone.htmlhead` is the first viewlet manager to be looked up like this, and it will fail firstly.

Some possible causes:

- You are trying to embed `main_template` inside form/view which is already rendered in `main_template` frame. Please see how to [embed forms and wrap forms manually](#).
- You might be using wrong `ViewPageTemplate` import (Five vs. `zope.pagetemplate` - explained elsewhere in this documentation)
- Make sure that you call `__of__()` method for views and other objects you construct by hand which expects themselves to be in the acquisition chain (normally discovered by traversing)

See also:

<https://bugs.launchpad.net/zope2/+bug/176566>

ERROR ZODB.Connection Couldn't load state for 0x00

Traceback:

```
2010-07-14 05:02:33 ERROR ZODB.Connection Couldn't load state for 0x00
Traceback (most recent call last):
  File "/Users/moo/yourinstance/eggs/ZODB3-3.8.4-py2.4-macosx-10.6-i386.egg/ZODB/
↳ Connection.py", line 811, in setstate
    self._setstate(obj)
  File "/Users/moo/yourinstance/eggs/ZODB3-3.8.4-py2.4-macosx-10.6-i386.egg/ZODB/
↳ Connection.py", line 870, in _setstate
    self._reader.setGhostState(obj, p)
  File "/Users/moo/yourinstance/eggs/ZODB3-3.8.4-py2.4-macosx-10.6-i386.egg/ZODB/
↳ serialize.py", line 604, in setGhostState
    state = self.getState(pickle)
  File "/Users/moo/yourinstance/eggs/ZODB3-3.8.4-py2.4-macosx-10.6-i386.egg/ZODB/
↳ serialize.py", line 597, in getState
    return unpickler.load()
  File "/Users/moo/yourinstance/eggs/ZODB3-3.8.4-py2.4-macosx-10.6-i386.egg/ZODB/
↳ serialize.py", line 471, in _persistent_load
    return self.load_oid(reference)
  File "/Users/moo/yourinstance/eggs/ZODB3-3.8.4-py2.4-macosx-10.6-i386.egg/ZODB/
↳ serialize.py", line 537, in load_oid
    return self._conn.get(oid)
  File "/Users/moo/yourinstance/eggs/ZODB3-3.8.4-py2.4-macosx-10.6-i386.egg/ZODB/
↳ Connection.py", line 244, in get
    p, serial = self._storage.load(oid, self._version)
  File "/Users/moo/yourinstance/eggs/ZODB3-3.8.4-py2.4-macosx-10.6-i386.egg/ZODB/
↳ FileStorage/FileStorage.py", line 470, in load
    pos = self._lookup_pos(oid)
  File "/Users/moo/yourinstance/eggs/ZODB3-3.8.4-py2.4-macosx-10.6-i386.egg/ZODB/
↳ FileStorage/FileStorage.py", line 462, in _lookup_pos
    raise POSKeyError(oid)
POSKeyError: 0x01
```

Condition: This error can happen when you try to start Zope

Reason: Data.fs might have been damaged. You might be using blobs with Plone 3 and they don't work perfectly. . . or a bunch other issues which generally mean that your day is screwed.

See also:

<http://plonechix.blogspot.com/2009/12/definitive-guide-to-poskeyerror.html>

Error _restore_index() when starting instance / ZEO server

Traceback:

```
2011-05-09 09:42:20 INFO ZServer HTTP server started at Mon May 9 09:42:20 2011
    Hostname: 0.0.0.0
    Port: 10997
2011-05-09 09:42:21 INFO Marshall libxml2-python not available. Unable to register_
↳ libxml2 based marshallers, at least SimpleXMLMarshaller
2011-05-09 09:42:22 INFO DocFinderTab Applied patch version 1.0.4.
Traceback (most recent call last):
```

```
File "/home/moo/code/python2/parts/opt/lib/python2.4/pdb.py", line 1066, in main
    pdb._runscript(mainpyfile)
File "/home/moo/code/python2/parts/opt/lib/python2.4/pdb.py", line 991, in _
↳runscript
    self.run(statement, globals=globals_, locals=locals_)
File "/home/moo/code/python2/parts/opt/lib/python2.4/bdb.py", line 366, in run
    exec cmd in globals, locals
File "<string>", line 1, in ?
File "/home/moo/xxx/parts/zope2/lib/python/Zope2/Startup/run.py", line 56, in ?
    run()
File "/home/moo/xxx/parts/zope2/lib/python/Zope2/Startup/run.py", line 21, in run
    starter.prepare()
File "/home/moo/xxx/parts/zope2/lib/python/Zope2/Startup/__init__.py", line 102, in _
↳prepare
    self.startZope()
File "/home/moo/xxx/parts/zope2/lib/python/Zope2/Startup/__init__.py", line 278, in _
↳startZope
    Zope2.startup()
File "/home/moo/xxx/parts/zope2/lib/python/Zope2/__init__.py", line 47, in startup
    _startup()
File "/home/moo/xxx/parts/zope2/lib/python/Zope2/App/startup.py", line 59, in _
↳startup
    DB = dbtab.getDatabase('/', is_root=1)
File "/home/moo/xxx/parts/zope2/lib/python/Zope2/Startup/datatypes.py", line 280, _
↳in getDatabase
    db = factory.open(name, self.databases)
File "/home/moo/xxx/parts/zope2/lib/python/Zope2/Startup/datatypes.py", line 178, _
↳in open
    DB = self.createDB(database_name, databases)
File "/home/moo/xxx/parts/zope2/lib/python/Zope2/Startup/datatypes.py", line 175, _
↳in createDB
    return ZODBDatabase.open(self, databases)
File "/home/moo/xxx/parts/zope2/lib/python/ZODB/config.py", line 97, in open
    storage = section.storage.open()
File "/home/moo/xxx/parts/zope2/lib/python/ZODB/config.py", line 135, in open
    quota=self.config.quota)
File "/home/moo/xxx/parts/zope2/lib/python/ZODB/FileStorage/FileStorage.py", line _
↳154, in __init__
    r = self._restore_index()
File "/home/moo/xxx/parts/zope2/lib/python/ZODB/FileStorage/FileStorage.py", line _
↳365, in _restore_index
    index = info.get('index')
```

Reason: Data.fs.index is corrupted.

Solution: Remove Data.fs.index file. The index will be rebuilt on the launch.

Error: Incorrect padding

Traceback:

```
2012-02-06 16:52:25 ERROR Zope.SiteErrorLog 1328539945.430.234286547911 http://
↳localhost:9888/index_html
Traceback (innermost last):
  Module ZPublisher.Publish, line 110, in publish
  Module ZPublisher.BaseRequest, line 588, in traverse
  Module Products.PluggableAuthService.PluggableAuthService, line 233, in validate
```

```

Module Products.PluggableAuthService.PluggableAuthService, line 559, in _
↪extractUserIds
Module Products.PluggableAuthService.plugins.CookieAuthHelper, line 121, in _
↪extractCredentials
Module base64, line 321, in decodestring
Error: Incorrect padding

```

Condition: This error can happen when you try to access any Plone site URL

Reason: It means that your browser most likely tries to serve bad cookies / auth info to Zope.

Solution: Clear browser cache, cookies, etc.

Exception: Type name not specified in createObject

Traceback:

```

Module ZPublisher.Publish, line 119, in publish
Module ZPublisher.mapply, line 88, in mapply
Module ZPublisher.Publish, line 42, in call_object
Module Products.CMFFormController.FSControllerPythonScript, line 104, in __call__
Module Products.CMFFormController.Script, line 145, in __call__
Module Products.CMFCore.FSPythonScript, line 140, in __call__
Module Shared.DC.Scripts.Bindings, line 313, in __call__
Module Shared.DC.Scripts.Bindings, line 350, in _bindAndExec
Module Products.CMFCore.FSPythonScript, line 196, in _exec
Module None, line 11, in createObject
<FSControllerPythonScript at /xxx/createObject used for /xxx/sisalto/lomapalvelut>
Line 11
Exception: Type name not specified

```

ExpatError: portlets.xml: unbound prefix

Traceback:

```

Traceback (innermost last):
Module plone.postpublicationhook.hook, line 74, in publish
Module ZPublisher.mapply, line 88, in mapply
Module ZPublisher.Publish, line 42, in call_object
Module Products.CMFQuickInstallerTool.QuickInstallerTool, line 589, in _
↪installProducts
Module Products.CMFQuickInstallerTool.QuickInstallerTool, line 526, in _
↪installProduct
- __traceback_info__: ('mfabrik.app',)
Module Products.GenericSetup.tool, line 390, in runAllImportStepsFromProfile
- __traceback_info__: profile-mfabrik.app:default
Module Products.GenericSetup.tool, line 1179, in _runImportStepsFromContext
Module Products.GenericSetup.tool, line 1090, in _doRunImportStep
- __traceback_info__: portlets
Module plone.app.portlets.exportimport.portlets, line 707, in importPortlets
Module Products.GenericSetup.utils, line 543, in _importBody
ExpatError: portlets.xml: unbound prefix: line 15, column 1

```

Condition: This error can happen while installing a new portlet portlets.xml

Reason: You have `i18n:attributes="title; description"` in your portlets.xml.

Solution: Remove it or declare the i18n namespace in XML like this:

```
<portlets xmlns:i18n="http://namespaces.zope.org/i18n">
```

Similar applies for actions.xml, etc.

IOError: [Errno url error] unknown url type: 'https'

Traceback:

```
File "/home/moo/code/python/parts/opt/lib/python2.4/urllib.py", line 89, in _
↳ urlretrieve
    return _urlopener.retrieve(url, filename, reporthook, data)
File "/home/moo/code/python/parts/opt/lib/python2.4/urllib.py", line 222, in retrieve
    fp = self.open(url, data)
File "/home/moo/code/python/parts/opt/lib/python2.4/urllib.py", line 187, in open
    return self.open_unknown(fullurl, data)
File "/home/moo/code/python/parts/opt/lib/python2.4/urllib.py", line 199, in open_
↳ unknown
    raise IOError, ('url error', 'unknown url type', type)
IOError: [Errno url error] unknown url type: 'https'
```

Reason: Python and Python socket modules have not been compiled with SSL support.

Solution: Make sure that you have SSL development libraries installed (Ubuntu/Debian example)

```
sudo apt-get install libssl-dev
```

Make sure that Python is built with SSL support

```
./configure --with-package=_ssl
```

You can test Python after compilation:

```
moo@murskaamo:~/code/python$ source python-2.4/bin/activate
(python-2.4)moo@murskaamo:~/code/python$ python
Python 2.4.6 (#1, Jul 16 2010, 10:31:46)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import _ssl
>>>
```

Also you might want try

```
easy_install pyopenssl
```

ImportError: Couldn't import ZPublisherEventsBackport

The following traceback on instance start-up:

```
File "/Users/moo/twinapex/parts/zope2/lib/python/zope/configuration/config.py", line _
↳ 1383, in toargs
    args[str(name)] = field.fromUnicode(s)
File "/Users/moo/twinapex/parts/zope2/lib/python/zope/configuration/fields.py", line _
↳ 141, in fromUnicode
    raise schema.ValidationError(v)
```

```

zope.configuration.xmlconfig.ZopeXMLConfigurationError: File "/Users/moo/twinapex/
↳ parts/instance/etc/site.zcml", line 14.2-14.55
    ZopeXMLConfigurationError: File "/Users/moo/twinapex/parts/instance/etc/package-
↳ includes/009-gomobile.mobile-configure.zcml", line 1.0-1.59
        ZopeXMLConfigurationError: File "/Users/moo/twinapex/src/gomobile.mobile/gomobile/
↳ mobile/configure.zcml", line 15.4-15.51
            ZopeXMLConfigurationError: File "/Users/moo/twinapex/eggs/plone.
↳ postpublicationhook-1.1-py2.4.egg/plone/postpublicationhook/configure.zcml", line 5.
↳ 4-8.10
                ConfigurationError: ('Invalid value for', 'package', "ImportError: Couldn't
↳ import ZPublisherEventsBackport, No module named ZPublisherEventsBackport")

```

Reason: plone.postpublicationhook 1.1 depends on new package, ZPublisherEventsBackport, for Plone 3.3.

Solution: You need to include them both in your buildout. You need to include both eggs:

```

eggs =
    ZPublisherEventsBackport
    plone.postpublicationhook

```

ImportError: Inappropriate file type for dynamic loading

Traceback:

```

File "/Users/moo/twinapex/twinapex/parts/zope2/lib/python/ZConfig/datatypes.py", line
↳ 398, in get
    t = self.search(name)
File "/Users/moo/twinapex/twinapex/parts/zope2/lib/python/ZConfig/datatypes.py", line
↳ 423, in search
    package = __import__(n, g, g, component)
File "/Users/moo/twinapex/twinapex/parts/zope2/lib/python/Zope2/Startup/datatypes.py",
↳ line 20, in ?
    from ZODB.config import ZODBDatabase
File "/Users/moo/twinapex/twinapex/eggs/ZODB3-3.8.2-py2.4-macosx-10.6-i386.egg/ZODB/___
↳ init__.py", line 20, in ?
    from persistent import TimeStamp
File "/Users/moo/twinapex/twinapex/eggs/ZODB3-3.8.2-py2.4-macosx-10.6-i386.egg/
↳ persistent/___init__.py", line 19, in ?
    from cPersistence import Persistent, GHOST, UPTODATE, CHANGED, STICKY
ImportError: Inappropriate file type for dynamic loading

```

Condition: When starting Zope

Reason: You probably have files lying over from wrong CPU architecture

- Hand copied eggs between servers
- Migrated OS to new version
- You have several Python interpreters installed and you try to run Zope using the wrong interpreter (the one which the code is not compiled for)

Solution: Delete /parts and /eggs buildout folders, run bootstrap, run buildout.

ImportError: No module named PIL

Traceback:

```

...
Traceback (most recent call last):
  File "/home/moo/isleofback/bin/idelauncher.py", line 140, in ?
    execfile(ZOPE_RUN)
  File "/home/moo/isleofback/bin/../parts/zope2/lib/python/Zope2/Startup/run.py",
↪line 56, in ?
    run()
  File "/home/moo/isleofback/bin/../parts/zope2/lib/python/Zope2/Startup/run.py",
↪line 21, in run
    starter.prepare()
  File "/home/moo/isleofback/parts/zope2/lib/python/Zope2/Startup/__init__.py", line
↪102, in prepare
    self.startZope()
  File "/home/moo/isleofback/parts/zope2/lib/python/Zope2/Startup/__init__.py", line
↪278, in startZope
    Zope2.startup()
  File "/home/moo/isleofback/parts/zope2/lib/python/Zope2/__init__.py", line 47, in
↪startup
    _startup()
  File "/home/moo/isleofback/parts/zope2/lib/python/Zope2/App/startup.py", line 45,
↪in startup
    OFS.Application.import_products()
  File "/home/moo/isleofback/parts/zope2/lib/python/OFS/Application.py", line 686, in
↪import_products
    import_product(product_dir, product_name, raise_exc=debug_mode)
  File "/home/moo/isleofback/parts/zope2/lib/python/OFS/Application.py", line 709, in
↪import_product
    product=__import__(pname, global_dict, global_dict, silly)
  File "/home/moo/isleofback/eggs/Products.ATContentTypes-1.3.4-py2.4.egg/Products/
↪ATContentTypes/__init__.py", line 64, in ?
    import Products.ATContentTypes.content
  File "/home/moo/isleofback/eggs/Products.ATContentTypes-1.3.4-py2.4.egg/Products/
↪ATContentTypes/content/__init__.py", line 26, in ?
    import Products.ATContentTypes.content.link
  File "/home/moo/isleofback/eggs/Products.ATContentTypes-1.3.4-py2.4.egg/Products/
↪ATContentTypes/content/link.py", line 39, in ?
    from Products.ATContentTypes.content.base import registerATCT
  File "/home/moo/isleofback/eggs/Products.ATContentTypes-1.3.4-py2.4.egg/Products/
↪ATContentTypes/content/base.py", line 63, in ?
    from Products.CMFPlone.PloneFolder import ReplaceableWrapper
  File "/home/moo/isleofback/eggs/Plone-3.3.5-py2.4.egg/Products/CMFPlone/__init__.py
↪", line 215, in ?
    from browser import ploneview
  File "/home/moo/isleofback/eggs/Plone-3.3.5-py2.4.egg/Products/CMFPlone/browser/
↪ploneview.py", line 12, in ?
    from Products.CMFPlone import utils
  File "/home/moo/isleofback/eggs/Plone-3.3.5-py2.4.egg/Products/CMFPlone/utils.py",
↪line 6, in ?
    from PIL import Image
ImportError: No module named PIL

```

Reason: Python Imaging Library is not properly installed. The default PIL package does not work nicely as egg.

Solution: Remove all existing PIL eggs from buildout/eggs folder.

Install PIL for your development Python environment:

```
easy_install http://dist.repoze.org/PIL-1.1.6.tar.gz
```

ImportError: No module named html

Traceback:

```

from lxml.html import defs
zope.configuration.xmlconfig.ZopeXMLConfigurationError: File "/srv/plone/yourinstance/
↳ parts/client1/etc/site.zcml", line 14.2-14.55
ZopeXMLConfigurationError: File "/srv/plone/yourinstance/parts/client1/etc/package-
↳ includes/012-yourinstance.mobi-configure.zcml", line 1.0-1.59
ZopeXMLConfigurationError: File "/srv/plone/yourinstance/src/yourinstance.mobi/
↳ yourinstance/mobi/configure.zcml", line 13.2-13.43
ZopeXMLConfigurationError: File "/srv/plone/yourinstance/src/gomobiletheme.basic/
↳ gomobiletheme/basic/configure.zcml", line 16.2-16.39
ZopeXMLConfigurationError: File "/srv/plone/yourinstance/src/gomobile.mobile/gomobile/
↳ mobile/configure.zcml", line 19.4-19.34
ZopeXMLConfigurationError: File "/srv/plone/yourinstance/src/gomobile.mobile/gomobile/
↳ mobile/browser/configure.zcml", line 24.4-29.10
ImportError: No module named html

```

Condition: This error can happen when starting an instance

Reason: The system lxml version is too old

Let's see if we are getting too old system wide lxml installation:

```

plone@mansikki:/srv/plone/yourinstance$ python2.4
Python 2.4.5 (#2, Jan 21 2010, 20:05:55)
[GCC 4.2.4 (Ubuntu 4.2.4-1ubuntu3)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import lxml
>>> lxml.__file__
'/usr/lib/python2.4/site-packages/lxml/__init__.pyc'
>>> dir(lxml)
['__builtins__', '__doc__', '__file__', '__name__', '__path__']
>>> from lxml import html
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ImportError: cannot import name html

```

If we cannot fix the system lxml (your system software depends on it) the only workaround is to create virtualenv. We cannot force Python 2.6, 2.5 or 2.4 not to use system libraries.

Example:

```

root@mansikki:/srv/plone# virtualenv -p /usr/bin/python2.4 --no-site-packages py24

```

Include standalone lxml + libxml compilation in your buildout.cfg:

```

parts =
    ...
    lxml

[lxml]
recipe = z3c.recipe.staticlxml
egg = lxml==2.2.6
force = false

```

If there are exiting lxml builds in buildout be sure they are removed:

```
rm -rf eggs/lxml*
```

Then as the non-root re-bootstrap the buildout using non-system wide Python:

```
plone@mansikki:/srv/plone/yourinstance-2010/yourinstance$ source /srv/plone/py24/bin/
↪activate
(py24)plone@mansikki:/srv/plone/yourinstance-2010/yourinstance$ python bootstrap.py
...
(py24)plone@mansikki:/srv/plone/yourinstance-2010/yourinstance$ bin/buildout
...
```

... and after this it should no longer pull the bad system lxml.

ImportError: No module named pkgutil

Traceback:

```
Traceback (most recent call last):
  File "/Users/moo/plonecommunity/bin/idelauncher.py", line 101, in <module>
    exec(data, globals())
  File "<string>", line 543, in <module>
  File "/Users/moo/plonecommunity/eggs/plone.app.z3cform-0.5.0-py2.6.egg/plone/__init_
↪_.py", line 5, in <module>
    from pkgutil import extend_path
ImportError: No module named pkgutil
```

If you are using Eclipse, idelauncher.py has been updated for Plone 4.

Invalid or Duplicate property id

Traceback:

```
* Dry run selected.
* Starting the migration from version: 3.1.4
* Attempting to upgrade from: 3.1.4
* Upgrade aborted
* Error type: zExceptions.BadRequest
* Error value: Invalid or duplicate property id
* File
"/usr/local/Plone3.2.3/buildout-cache/eggs/Plone-3.3-py2.4.egg/Products/CMFPlone/
↪MigrationTool.py",
line 210, in upgrade newv, msgs = self._upgrade(newv)
  * File
"/usr/local/Plone3.2.3/buildout-cache/eggs/Plone-3.3-py2.4.egg/Products/CMFPlone/
↪MigrationTool.py",
line 321, in _upgrade res = function(self.aq_parent)
  * File
"/usr/local/Plone3.2.3/buildout-cache/eggs/Plone-3.3-py2.4.egg/Products/CMFPlone/
↪migrations/v3_1/final_three1x.py",
line 15, in three14_three15 loadMigrationProfile(portal,
'profile-Products.CMFPlone.migrations:3.1.3-3.1.4')
  * File
"/usr/local/Plone3.2.3/buildout-cache/eggs/Plone-3.3-py2.4.egg/Products/CMFPlone/
↪migrations/migration_util.py",
line 107, in loadMigrationProfile tool.runAllImportStepsFromProfile(profile,
purge_old=False)
```



```

* File
"/usr/local/Plone3.2.3/buildout-cache/eggs/Products.GenericSetup-1.4.5-py2.4.egg/
↳Products/GenericSetup/tool.py",
line 390, in runAllImportStepsFromProfile
ignore_dependencies=ignore_dependencies)
* File
"/usr/local/Plone3.2.3/buildout-cache/eggs/Products.GenericSetup-1.4.5-py2.4.egg/
↳Products/GenericSetup/tool.py",
line 1179, in _runImportStepsFromContext message =
self._doRunImportStep(step, context)
* File
"/usr/local/Plone3.2.3/buildout-cache/eggs/Products.GenericSetup-1.4.5-py2.4.egg/
↳Products/GenericSetup/tool.py",
line 1090, in _doRunImportStep return handler(context)
* File
"/usr/local/Plone3.2.3/buildout-cache/eggs/Plone-3.3-py2.4.egg/Products/CMFPlone/
↳exportimport/propertiestool.py",
line 37, in importPloneProperties importer.body = body
* File
"/usr/local/Plone3.2.3/buildout-cache/eggs/Products.GenericSetup-1.4.5-py2.4.egg/
↳Products/GenericSetup/utils.py",
line 544, in _importBody self._importNode(dom.documentElement)
* File
"/usr/local/Plone3.2.3/buildout-cache/eggs/Plone-3.3-py2.4.egg/Products/CMFPlone/
↳exportimport/propertiestool.py",
line 103, in _importNode self._initObjects(node)
* File
"/usr/local/Plone3.2.3/buildout-cache/eggs/Plone-3.3-py2.4.egg/Products/CMFPlone/
↳exportimport/propertiestool.py",
line 154, in _initObjects importer.node = child
* File
"/usr/local/Plone3.2.3/buildout-cache/eggs/Plone-3.3-py2.4.egg/Products/CMFPlone/
↳exportimport/propertiestool.py",
line 77, in _importNode self._initProperties(node)
* File
"/usr/local/Plone3.2.3/buildout-cache/eggs/Products.GenericSetup-1.4.5-py2.4.egg/
↳Products/GenericSetup/utils.py",
line 724, in _initProperties obj._setProperty(prop_id, val, prop_type)
* File
"/usr/local/Plone3.2.3/Zope-2.10.7-final-py2.4/lib/python/OFS/PropertyManager.py",
line 186, in _setProperty raise BadRequest, 'Invalid or duplicate property
id'

* End of upgrade path, migration has finished
* The upgrade path did NOT reach current version
* Migration has failed
* Dry run selected, transaction aborted

```

Condition: This exception can happen during Plone migration to the newer version

It is caused by a property (site setting) which already exists and migration tries to create it. The usual reason is that one has edited site settings in new Plone version before running the migration.

Try remove violating property ids from the site_properties manually in Zope.

Potential candidates to be removed:

- enable_inline_editing
- lock_on_ttw_edit (boolean)

Potential candidates which need to be added manually:

- `redirect_links` (boolean)

See also:

<http://www.mail-archive.com/setup@lists.plone.org/msg03988.html>

InvalidInterface: Concrete attribute

Traceback:

```
/zope/interface/interface.py", line 495, in __init__
    raise InvalidInterface("Concrete attribute, " + name)
zope.configuration.xmlconfig.ZopeXMLConfigurationError: File "/Users/mikko/code/
↳ buildout.deco/parts/instance/etc/site.zcml", line 15.2-15.55
    ZopeXMLConfigurationError: File "/Users/mikko/code/buildout.deco/parts/instance/
↳ etc/package-includes/002-plone.app.widgets-configure.zcml", line 1.0-1.61
    ZopeXMLConfigurationError: File "/Users/mikko/code/buildout.deco/src/plone.app.
↳ widgets/plone/app/widgets/configure.zcml", line 56.2-62.6
    InvalidInterface: Concrete attribute, multiChoiceCheckbox
```

Condition: Your `zope.schema` based schema breaks on Plone startup.

Reason: You have extra comma in your schema. Like this:

```
class IChoiceExamples(model.Schema):

    multiChoiceCheckbox = zope.schema.List(
        title=u"Checkbox multiple choices",
        description=u"Select multiple checkboxes using checkboxes and store_
↳ values in zope.schema.List (maps to python List)." + DEFAULT_MUTABLE_WARNING,
        required=False,
        value_type=zope.schema.Choice(vocabulary="plone.app.vocabularies.
↳ PortalTypes")), # <---- This is the guilty comma
```

Iteration over non-sequence in `_normalizeargs`

Case 1

The following log trace will appear when you try to render the site, but you can access the Management Interface normally:

```
2009-09-23 20:47:18 WARNING OFS.Uninstalled Could not import class
↳ 'IPloneCommentsLayer' from module 'quintagroup.plonecomments.interfaces'
2009-09-23 20:47:18 ERROR Zope.SiteErrorLog 1253728038.160.534632167217 http://
↳ localhost:9444/XXX
Traceback (innermost last):
  Module plone.postpublicationhook.hook, line 65, in publish
  Module ZPublisher.BaseRequest, line 424, in traverse
  Module ZPublisher.BeforeTraverse, line 99, in __call__
  Module Products.CMFCore.PortalObject, line 94, in __before_publishing_traverse__
  Module zope.event, line 23, in notify
  Module zope.component.event, line 26, in dispatch
  Module zope.component._api, line 130, in subscribers
  Module zope.component.registry, line 290, in subscribers
  Module zope.interface.adapter, line 535, in subscribers
```

```

Module zope.component.event, line 33, in objectEventNotify
Module zope.component._api, line 130, in subscribers
Module zope.component.registry, line 290, in subscribers
Module zope.interface.adapter, line 535, in subscribers
Module plone.browserlayer.layer, line 18, in mark_layer
Module zope.interface.declarations, line 848, in directlyProvides
Module zope.interface.declarations, line 1371, in _normalizeargs
Module zope.interface.declarations, line 1370, in _normalizeargs
TypeError: iteration over non-sequence
2009-09-23 20:47:18 ERROR root Exception while rendering an error message
Traceback (most recent call last):
  File "/home/moo/XXX/parts/zope2/lib/python/OFS/SimpleItem.py", line 227, in raise_
↳ standardErrorMessage
    v = s(**kwargs)
  File "/home/moo/workspace2/collective.skinny/collective/skinny/patch.py", line 8,
↳ in standard_error_message
    return self.restrictedTraverse('@@404.html')()
  File "/home/moo/workspace2/collective.skinny/collective/skinny/fourohfour.py", line
↳ 22, in __call__
    return skins.plone_templates.standard_error_message.__of__(
  File "/home/moo/XXX/eggs/Products.CMFCore-2.1.2-py2.4.egg/Products/CMFCore/
↳ FSPythonScript.py", line 140, in __call__
    return Script.__call__(self, *args, **kw)
  File "/home/moo/XXX/parts/zope2/lib/python/Shared/DC/Scripts/Bindings.py", line 313,
↳ in __call__
    return self._bindAndExec(args, kw, None)
  File "/home/moo/XXX/parts/zope2/lib/python/Shared/DC/Scripts/Bindings.py", line 350,
↳ in _bindAndExec
    return self._exec(bound_data, args, kw)
  File "/home/moo/XXX/eggs/Products.CMFCore-2.1.2-py2.4.egg/Products/CMFCore/
↳ FSPythonScript.py", line 196, in _exec
    result = f(*args, **kw)
  File "Script (Python)", line 27, in standard_error_message
AttributeError: default_error_message

```

This usually means that you have copied Data.fs from another system, but you do not have identical add-on product configuration installed.

traceback to the console similar to the following if you have started Zope process on foreground:

```

2008-11-09 22:53:13 INFO Zope Ready to handle requests
2008-11-09 22:54:50 WARNING OFS.Uninstalled Could not import class 'ATSETemplateTool'
↳ from module 'Products.ATSchemaEditorNG.ATSETemplateTool'
2008-11-09 22:54:50 WARNING OFS.Uninstalled Could not import class 'SchemaEditorTool'
↳ from module 'Products.ATSchemaEditorNG.SchemaEditorTool'
2008-11-09 22:54:50 WARNING OFS.Uninstalled Could not import class 'SchemaManagerTool'
↳ from module 'Products.GenericPloneContent.SchemaManagerTool'
2008-11-09 22:54:50 WARNING OFS.Uninstalled Could not import class 'FormGenTool' from
↳ module 'Products.PloneFormGen.tools.formGenTool'
2008-11-09 22:54:50 WARNING OFS.Uninstalled Could not import class 'TemplatedDocument'
↳ from module 'collective.easytemplate.content.TemplatedDocument'
2008-11-09 22:54:50 WARNING OFS.Uninstalled Could not import class 'FormFolder' from
↳ module 'Products.PloneFormGen.content.form'
2008-11-09 22:54:52 WARNING OFS.Uninstalled Could not import class 'IDropdownSpecific'
↳ from module 'webcouturier.dropdownmenu.browser.interfaces'
2008-11-09 22:54:52 ERROR Zope.SiteErrorLog http://localhost:8080/lsm
Traceback (innermost last):
  Module ZPublisher.Publish, line 110, in publish

```

```
Module ZPublisher.BaseRequest, line 424, in traverse
Module ZPublisher.BeforeTraverse, line 99, in __call__
Module Products.CMFCore.PortalObject, line 94, in __before_publishing_traverse__
Module zope.event, line 23, in notify
Module zope.component.event, line 26, in dispatch
Module zope.component._api, line 130, in subscribers
Module zope.component.registry, line 290, in subscribers
Module zope.interface.adapter, line 535, in subscribers
Module zope.component.event, line 33, in objectEventNotify
Module zope.component._api, line 130, in subscribers
Module zope.component.registry, line 290, in subscribers
Module zope.interface.adapter, line 535, in subscribers
Module plone.browserlayer.layer, line 18, in mark_layer
Module zope.interface.declarations, line 848, in directlyProvides
Module zope.interface.declarations, line 1371, in _normalizeargs
Module zope.interface.declarations, line 1370, in _normalizeargs
TypeError: iteration over non-sequence
```

notice the ‘Could not import class’ message.

Reason: You do not have identical product configuration on the new server. Please install the missing products and site should work fine again.

Please note that you can get a ‘TypeError: iteration over non-sequence’ exception in other contexts not related with missing products at all. Look for the ‘Could not import class’ message in your traceback.

Case 2

Example traceback:

```
Traceback (most recent call last):
  File "/home/moo/twinapex/bin/idelauncher.py", line 158, in ?
    execfile(ZOPE_RUN)
  File "/home/moo/twinapex/bin/./parts/zope2/lib/python/Zope2/Startup/run.py", line
↪56, in ?
    run()
  File "/home/moo/twinapex/bin/./parts/zope2/lib/python/Zope2/Startup/run.py", line
↪21, in run
    starter.prepare()
  File "/home/moo/twinapex/parts/zope2/lib/python/Zope2/Startup/__init__.py", line
↪102, in prepare
    self.startZope()
  File "/home/moo/twinapex/parts/zope2/lib/python/Zope2/Startup/__init__.py", line
↪278, in startZope
    Zope2.startup()
  File "/home/moo/twinapex/parts/zope2/lib/python/Zope2/__init__.py", line 47, in
↪startup
    _startup()
  File "/home/moo/twinapex/parts/zope2/lib/python/Zope2/App/startup.py", line 45, in
↪startup
    OFS.Application.import_products()
  File "/home/moo/twinapex/parts/zope2/lib/python/OFS/Application.py", line 686, in
↪import_products
    import_product(product_dir, product_name, raise_exc=debug_mode)
  File "/home/moo/twinapex/parts/zope2/lib/python/OFS/Application.py", line 709, in
↪import_product
    product=__import__(pname, global_dict, global_dict, silly)
```

```

File "/home/moo/twinapex/eggs/Products.PloneHelpCenter-4.0a1-py2.4.egg/Products/
↳ PloneHelpCenter/__init__.py", line 9, in ?
    from Products.PloneHelpCenter import content
File "/home/moo/twinapex/eggs/Products.PloneHelpCenter-4.0a1-py2.4.egg/Products/
↳ PloneHelpCenter/content/__init__.py", line 10, in ?
    import HowToFolder, HowTo
File "/home/moo/twinapex/eggs/Products.PloneHelpCenter-4.0a1-py2.4.egg/Products/
↳ PloneHelpCenter/content/HowTo.py", line 40, in ?
    class HelpCenterHowTo(PHCCContentMixin, ATCTOrderedFolder):
File "/home/moo/twinapex/parts/zope2/lib/python/zope/interface/advice.py", line 132,
↳ in advise
    return callback(newClass)
File "/home/moo/twinapex/parts/zope2/lib/python/zope/interface/declarations.py",
↳ line 485, in _implements_advice
    classImplements(cls, *interfaces)
File "/home/moo/twinapex/parts/zope2/lib/python/zope/interface/declarations.py",
↳ line 462, in classImplements
    spec.declared += tuple(_normalizeargs(interfaces))
File "/home/moo/twinapex/parts/zope2/lib/python/zope/interface/declarations.py",
↳ line 1372, in _normalizeargs
    _normalizeargs(v, output)
File "/home/moo/twinapex/parts/zope2/lib/python/zope/interface/declarations.py",
↳ line 1371, in _normalizeargs
    for v in sequence:
TypeError: iteration over non-sequence

```

Reason: You are trying to use Plone 4 (Zope 2.12) add-on on Plone 3 (Zope 2.10). Zope interface declarations have been changed.

Solution 1: Pick the older version for the add-on which is known to work with Plone 3. Make sure that you delete all “too eggs” from eggs/ and src/ folders.

Solution 2: Upgrade your site to Plone.

NameError: name ‘test’ is not defined

Condition: This exception occurs when you try to customize TAL page template code using test() function. test() function has been dropped in Zope 3 page templates. You should no longer use test() function anywhere.

Solution: replace test() with common Python expression in your customized template.

For example the original:

```

tal:attributes="class python:test(here.Format() in ('text/structured', 'text/x-rst',
↳ ), 'stx' + kss_class, 'plain', + kss_class)"

```

would need to be written as:

```

tal:attributes="class python:here.Format() in ('text/structured', 'text/x-rst', ) and
↳ 'stx' + kss_class or 'plain' + kss_class"

```

NotFound error (Page not found) when accessing @@manage-portlets

If you get *Page not found* error when accessing @@manage-portlets the first thing you need to do is to enable logging of NotFound exceptions in the Management Interface in error_log.

After that reload @@manage-portlets.

When you try to access @@manage-portlets an exception a NotFound exception is raised:

```

2009-11-09 12:56:13 ERROR Zope.SiteErrorLog 1257764173.180.738005333766 http://
↳localhost:8080/yourinstance/@@manage-portlets
Traceback (innermost last):
  Module ZPublisher.Publish, line 119, in publish
    Module Products.PageTemplates.Expressions, line 223, in evaluateStructure
    ...
  Module zope.tales.tales, line 696, in evaluate
    - URL: file:/Users/moo/workspace/plonetheme.yourinstance/plonetheme/yourinstance/
↳skins/plonetheme_yourinstance_custom_templates/main_template.pt
    - Line 92, Column 18
    - Expression: <StringExpr u'plone.leftcolumn'>
    - Names:
      {'container': <PloneSite at /yourinstance>,
       'context': <PloneSite at /yourinstance>,
       'default': <object object at 0x194520>,
       'here': <PloneSite at /yourinstance>,
       'loop': {},
       'nothing': None,
       'options': {'args': (<Products.Five.metaclass.SimpleViewClass from /Users/moo/
↳yourinstance/eggs/plone.app.portlets-1.2-py2.4.egg/plone/app/portlets/browser/
↳templates/manage-contextual.pt object at 0x67e43b0>,)},
       'repeat': <Products.PageTemplates.Expressions.SafeMapping object at 0x73b59b8>,
       'request': <HTTPRequest, URL=http://localhost:8080/yourinstance/@@manage-
↳portlets>,
       'root': <Application at >,
       'template': <ImplicitAcquirerWrapper object at 0x73b29f0>,
       'traverse_subpath': [],
       'user': <PropertiedUser 'admin'>,
       'view': <Products.Five.metaclass.SimpleViewClass from /Users/moo/yourinstance/
↳eggs/plone.app.portlets-1.2-py2.4.egg/plone/app/portlets/browser/templates/manage-
↳contextual.pt object at 0x67e43b0>,
       'views': <zope.app.pagetemplate.viewpagetemplatefile.ViewMapper object at_
↳0x73b23d0>}
  Module Products.Five.browser.providerexpression, line 37, in __call__
    ...
  Module zope.tales.tales, line 696, in evaluate
    - URL: index
    - Line 18, Column 12
    - Expression: <PathExpr standard:'view/addable_portlets'>
    - Names:
      {'container': <PloneSite at /yourinstance>,
       'context': <PloneSite at /yourinstance>,
       'default': <object object at 0x194520>,
       'here': <PloneSite at /yourinstance>,
       'loop': {},
       'nothing': None,
       'options': {'args': ()},
       'repeat': <Products.PageTemplates.Expressions.SafeMapping object at 0x7941be8>,
       'request': <HTTPRequest, URL=http://localhost:8080/yourinstance/@@manage-
↳portlets>,
       'root': <Application at >,
       'template': <ImplicitAcquirerWrapper object at 0x78be050>,
       'traverse_subpath': [],
       'user': <PropertiedUser 'admin'>,
       'view': <plone.app.portlets.browser.editmanager.
↳ContextualEditPortletManagerRenderer object at 0x789eb90>,
       'views': <zope.app.pagetemplate.viewpagetemplatefile.ViewMapper object at_
↳0x790a870>}

```

```

Module zope.tales.expressions, line 217, in __call__
Module Products.PageTemplates.Expressions, line 163, in _eval
Module Products.PageTemplates.Expressions, line 125, in render
Module plone.app.portlets.browser.editmanager, line 154, in addable_portlets
Module plone.app.portlets.browser.editmanager, line 149, in check_permission
Module OFS.Traversable, line 301, in restrictedTraverse
Module OFS.Traversable, line 284, in unrestrictedTraverse
- __traceback_info__: ([], 'collective.easytemplate.TemplatedPortlet')
NotFound: collective.easytemplate.TemplatedPortlet

```

This usually means that your site has an portlet assignment which code is not present anymore.

In this case you can see that portlet type “collective.easytemplate.TemplatedPortlet” is missing.

Ä Check that you include the corresponding product (collective.easytemplate) in eggs= section in buildout.cfg

- Reinstall removed egg which has the code for the portlet
- Check that you include the corresponding product (collective.easytemplate) in zcml= section in buildout.cfg
- Make sure that portlet name is the same in ZCML and GenericSetup XML
- Make sure you use <include package=".portlets" /> in your code

Manually removing the portlet

If you have a traceback like this:

```

URL: index
Line 18, Column 12
Expression: <PathExpr standard:'view/addable_portlets'>
Names:
{'container': <ATFolder at /webandmobile/support>,
 'context': <ATFolder at /webandmobile/support>,
 'default': <object object at 0x7f7e3af1a200>,
 'here': <ATFolder at /webandmobile/support>,
 'loop': {},
 'nothing': None,
 'options': {'args': ()},
 'repeat': <Products.PageTemplates.Expressions.SafeMapping object at 0x11dee1b8>,
 'request': <HTTPRequest, URL=http://webandmobile.mfabrik.com/support/@@manage-
↳portlets>,
 'root': <Application at >,
 'template': <ImplicitAcquirerWrapper object at 0x7f7e2a9199d0>,
 'traverse_subpath': [],
 'user': <PropertiedUser 'admin'>,
 'view': <plone.app.portlets.browser.editmanager.ContextualEditPortletManagerRender_
↳object at 0xf0526d0>,
 'views': <zope.app.pagetemplate.viewpagetemplatefile.ViewMapper object at_
↳0x7f7e2a919810>}
Module zope.tales.expressions, line 217, in __call__
Module Products.PageTemplates.Expressions, line 163, in _eval
Module Products.PageTemplates.Expressions, line 125, in render
Module plone.app.portlets.browser.editmanager, line 154, in addable_portlets
Module plone.app.portlets.browser.editmanager, line 149, in check_permission
Module OFS.Traversable, line 301, in restrictedTraverse
Module OFS.Traversable, line 284, in unrestrictedTraverse
__traceback_info__: ([], 'gomobile.convergence.ContentMedia')
NotFound: gomobile.convergence.ContentMedia

```

It usually means that there is a portlet in your content which product code has been removed.

Reinstall the add-on providing the portlet, remove the portlet and then uninstall the add-on again.

NotFound while accessing a BrowserView based view

Traceback:

```
Traceback (innermost last):
  Module ZPublisher.Publish, line 110, in publish
  Module ZPublisher.BaseRequest, line 506, in traverse
  Module ZPublisher.HTTPResponse, line 686, in debugError
NotFound: <h2>Site Error</h2>
```

Condition: You'll get a NotFound error when accessing view using view traverse notation, even though the view exist.

Example URL:

```
http://yoursite/@@myview
```

Reason: This is because there is an exception raised in your view's `__init__()` method. Views are Zope multi-adapters. Exception in multi-adapter factory method causes `ComponentLookupError`. Zope 2 publisher translates this to NotFound error.

Solution: * Put *`pdb break statement`* to the beginning of the `__init__()` method of your view. Then step through view code to see where the exception is raised. * If your view does not have `__init__()` method, then copy the source code `__init__()` method to your view class from the first parent class which has a view

POSKeyError

POSKeyError is when the database has been unable to convert a reference to an object into the object itself It's a low level error usually caused by a corrupt or incomplete database.

- You did not copy blobs when you copied Data.fs
- Your data is corrupted
- Glitch in database (very unlikely)

See also:

<http://rpatterson.net/blog/poskeyerror-during-commit>

PicklingError: Can't pickle <class 'collective.singing.async.IQueue'>: import of module collective.singing.async

Singing & Dancing add-on does not uninstall cleanly. Try this command-line script to get it fixed (not tested). Some parts may work, some not, depending on how messed up your site is.

Note that you need to have S & D present in the buildout when running this and then you can remove it afterwards:

```
import transaction
from collective.singing.interfaces import ISalt
from collective.singing.async import IQueue
```



```
# Your site here
portal = app.mfabrik
sm = portal.getSiteManager()

util_obj = sm.getUtility(ISalt)
sm.unregisterUtility(provided=ISalt)
del util_obj

sm.utilities.unsubscribe((), ISalt)
del sm.utilities.__dict__['_provided'][ISalt]
del sm.utilities._subscribers[0][ISalt]

util = sm.queryUtility(IQueue, name='collective.dancing.jobs')
sm.unregisterUtility(util, IQueue, name='collective.dancing.jobs')
del util
del sm.utilities._subscribers[0][IQueue]

transaction.commit()
```

RuntimeError: maximum recursion depth exceeded (Archetypes field problem)

Traceback:

```
- __traceback_info__: ('memberimage', <TTMemberImage at tt_member_image.2010-01-23.
↳8138248069>, {'field': <Field memberimage(image:rw)>})
Module Products.Archetypes.Storage, line 96, in get
Module Products.Archetypes.utils, line 808, in shasattr
Module Products.Archetypes.fieldproperty, line 101, in __get__
Module Products.Archetypes.Field, line 997, in get
Module Products.Archetypes.Field, line 709, in get
- __traceback_info__: ('memberimage', <TTMemberImage at tt_member_image.2010-01-23.
↳8138248069>, {'field': <Field memberimage(image:rw)>})
RuntimeError: maximum recursion depth exceeded
```

Condition: The following code will generate this error when you try to access the object:

```
atapi.ImageField(
    'memberimage',
    # storage=atapi.AnnotationStorage(), # paster version
    storage=atapi.AttributeStorage(), # results in "max recursion depth exceeded"
↳error
    widget=atapi.ImageWidget(
        label=_("New Field"),
        description=_("Field description"),
    ),
    validators=('isNonEmptyFile'),
    original_size=(600,600),
    sizes={ 'mini' : (80,80),
            'normal' : (200,200),
            'big' : (300,300),
            'maxi' : (500,500)},
),
```

Reason: Schema fields using AttributeStorage (usually images, files) **cannot** have ATFieldProperty in the class:

```
class Sample(base.ATCTContent):

    # This does not work with AttributeStorage
    memberimage = atapi.ATFieldProperty('memberimage')
```

Solution: simply remove `ATFieldProperty()` declaration for the problematic field. You cannot access the field value anymore by calling `object.memberimage` but you need to call `object.getMemberimage()` instead.

TraversalError with lots of tuples and lists (METAL problem)

Traceback:

```
File "/home/moo/yourinstance/parts/zope2/lib/python/zope/tales/expressions.py", line 217, in __call__
    return self._eval(econtext)
File "/home/moo/yourinstance/parts/zope2/lib/python/Products/PageTemplates/Expressions.py", line 155, in _eval
    ob = self._subexprs[-1](econtext)
File "/home/moo/yourinstance/parts/zope2/lib/python/zope/tales/expressions.py", line 124, in _eval
    ob = self._traverser(ob, element, econtext)
File "/home/moo/yourinstance/parts/zope2/lib/python/Products/PageTemplates/Expressions.py", line 85, in boboAwareZopeTraverse
    request=request)
File "/home/moo/yourinstance/parts/zope2/lib/python/zope/traversing/adapters.py", line 164, in traversePathElement
    return traversable.traverse(nm, further_path)
- __traceback_info__: ({'main': [('version', '1.6'), ('mode', 'html'), ('setPosition', (7, 0)), ('setSourceFile', 'file:/home/moo/workspace2/collective.skinny/collective/skinny/skins/skinny_faux_layer/main_template.pt'), ('beginScope', {'define-macro': u'main'})], ('optTag', (u'metal:main-macro', None, 'metal', 0, [('startTag', (u'metal:main-macro', [(u'define-macro', u'main', 'metal'))]), [('rawtextColumn', (u'\n\t', 1)), ('setPosition', (8, 1)), ('defineSlot', (u'main', [('beginScope', {u'define-slot': u'main'})], ('optTag', (u'metal:main-slot', None, 'metal', 0, [('startTag', (u'metal:main-slot', [(u'define-slot', u'main', 'metal'))]), [('rawtextColumn', (u'\n\t', 1))]), ('endScope', ())), ('setPosition', (9, 1)), ('setSourceFile', 'file:/home/moo/workspace2/collective.skinny/collective/skinny/skins/skinny_faux_layer/main_template.pt'), ('rawtextColumn', (u'\n', 0)))]], ('endScope', ())), 'master'])
File "/home/moo/yourinstance/parts/zope2/lib/python/zope/traversing/adapters.py", line 52, in traverse
    raise TraversalError(subject, name)
- __traceback_info__: ({'main': [('version', '1.6'), ('mode', 'html'), ('setPosition', (7, 0)), ('setSourceFile', 'file:/home/moo/workspace2/collective.skinny/collective/skinny/skins/skinny_faux_layer/main_template.pt'), ('beginScope', {'define-macro': u'main'})], ('optTag', (u'metal:main-macro', None, 'metal', 0, [('startTag', (u'metal:main-macro', [(u'define-macro', u'main', 'metal'))]), [('rawtextColumn', (u'\n\t', 1)), ('setPosition', (8, 1)), ('defineSlot', (u'main', [('beginScope', {u'define-slot': u'main'})], ('optTag', (u'metal:main-slot', None, 'metal', 0, [('startTag', (u'metal:main-slot', [(u'define-slot', u'main', 'metal'))]), [('rawtextColumn', (u'\n\t', 1))]), ('endScope', ())), ('setPosition', (9, 1)), ('setSourceFile', 'file:/home/moo/workspace2/collective.skinny/collective/skinny/skins/skinny_faux_layer/main_template.pt'), ('rawtextColumn', (u'\n', 0)))]], ('endScope', ())), 'master', []])
TraversalError: ({'main': [('version', '1.6'), ('mode', 'html'), ('setPosition', (7, 0)), ('setSourceFile', 'file:/home/moo/workspace2/collective.skinny/collective/skinny/skins/skinny_faux_layer/main_template.pt'), ('beginScope', {'define-macro': u'main'})], ('optTag', (u'metal:main-macro', None, 'metal', 0, [('startTag', (u'metal:main-macro', [(u'define-macro', u'main', 'metal'))]), [('rawtextColumn', (u'\n\t', 1)), ('setPosition', (8, 1)), ('defineSlot', (u'main', [('beginScope', {u'define-slot': u'main'})], ('optTag', (u'metal:main-slot', None, 'metal', 0, [('startTag', (u'metal:main-slot', [(u'define-slot', u'main', 'metal'))]), [('rawtextColumn', (u'\n\t', 1))]), ('endScope', ())), ('setPosition', (9, 1)), ('setSourceFile', 'file:/home/moo/workspace2/collective.skinny/collective/skinny/skins/skinny_faux_layer/main_template.pt'), ('rawtextColumn', (u'\n', 0)))]], ('endScope', ())), 'master', []])
Chapter 5. Installing, Managing and Updating Plone
```

Some template tries to call macro inside another template and the macro is not defined in the target template.

TraversalError(subject, name) in expressions

Traceback:

```
File "/home/moo/sits/parts/zope2/lib/python/ZPublisher/Publish.py", line 119, in _
    publish
    request, bind=1)
File "/home/moo/sits/parts/zope2/lib/python/ZPublisher/mapply.py", line 88, in mapply
    if debug is not None: return debug(object, args, context)
File "/home/moo/sits/parts/zope2/lib/python/ZPublisher/Publish.py", line 42, in call_
    object
    result=apply(object, args) # Type s<cr> to step into published object.
File "/home/moo/sits/parts/zope2/lib/python/Products/Five/browser/metaconfigure.py",
    line 417, in __call__
    return self.index(self, *args, **kw)
File "/home/moo/sits/parts/zope2/lib/python/Shared/DC/Scripts/Bindings.py", line 313,
    in __call__
    return self._bindAndExec(args, kw, None)
File "/home/moo/sits/parts/zope2/lib/python/Shared/DC/Scripts/Bindings.py", line 350,
    in _bindAndExec
    return self._exec(bound_data, args, kw)
File "/home/moo/sits/parts/zope2/lib/python/Products/PageTemplates/PageTemplateFile.py
    ", line 129, in _exec
    return self.pt_render(extra_context=bound_names)
File "/home/moo/sits/parts/zope2/lib/python/Products/PageTemplates/PageTemplate.py",
    line 98, in pt_render
    showtal=showtal)
File "/home/moo/sits/parts/zope2/lib/python/zope/pagetemplate/pagetemplate.py", line
    117, in pt_render
    strictinsert=0, sourceAnnotations=sourceAnnotations)()
File "/home/moo/sits/parts/zope2/lib/python/zope/tal/talinterpreter.py", line 271, in
    __call__
    self.interpret(self.program)
File "/home/moo/sits/parts/zope2/lib/python/zope/tal/talinterpreter.py", line 346, in
    interpret
    handlers[opcode](self, args)
File "/home/moo/sits/parts/zope2/lib/python/zope/tal/talinterpreter.py", line 891, in
    do_useMacro
    self.interpret(macro)
    handlers[opcode](self, args)

...

File "/home/moo/sits/parts/zope2/lib/python/zope/tal/talinterpreter.py", line 586, in
    do_setLocal_tal
    self.engine.setLocal(name, self.engine.evaluateValue(expr))
File "/home/moo/sits/parts/zope2/lib/python/zope/tales/tales.py", line 696, in
    evaluate
    return expression(self)
File "/home/moo/sits/parts/zope2/lib/python/zope/tales/expressions.py", line 218, in
    __call__
    return self._eval(econtext)
File "/home/moo/sits/parts/zope2/lib/python/Products/PageTemplates/Expressions.py",
    line 153, in _eval
```

```
ob = self._subexprs[-1](econtext)
File "/home/moo/sits/parts/zope2/lib/python/zope/tales/expressions.py", line 124, in _
↳eval
ob = self._traverser(ob, element, econtext)
File "/home/moo/sits/parts/zope2/lib/python/Products/PageTemplates/Expressions.py",
↳line 103, in trustedBoboAwareZopeTraverse
request=request)
File "/home/moo/sits/parts/zope2/lib/python/zope/traversing/adapters.py", line 164,
↳in traversePathElement
return traversable.traverse(nm, further_path)
File "/home/moo/sits/parts/zope2/lib/python/zope/traversing/adapters.py", line 52, in
↳traverse
raise TraversalError(subject, name)
```

Reason: From line Products/PageTemplates/Expressions.py you can see the error comes from TAL templates. TAL templates are trying to execute path based expressions.

If you can view this error through error_log the error_log traceback will contain information what expression causes the exception. However if this only happens with unit tests you can have something like:

```
def __call__(self, econtext):
    if self._name == 'exists':
        return self._exists(econtext)
    print "Evaluating expression:" + self._s
    return self._eval(econtext)
```

manually injected to zope.tales.expression module.

TraversalError: @@standard_macros

Traceback:

```
- Warning: Macro expansion failed
- Warning: zope.traversing.interfaces.TraversalError: (<plone.app.headeranimation.
↳browser.forms.HeaderCRUDForm object at 0x110289590>, '++view++standard_macros')
Module zope.tal.talinterpreter, line 271, in __call__
Module zope.tal.talinterpreter, line 346, in interpret
Module zope.tal.talinterpreter, line 870, in do_useMacro
Module zope.tales.tales, line 696, in evaluate
- URL: form
- Line 1, Column 0
- Expression: <PathExpr standard:'context/@@standard_macros/page'>
- Names:
  {'container': <plone.app.headeranimation.browser.forms.HeaderCRUDForm object at
↳0x110289590>,
  'context': <plone.app.headeranimation.browser.forms.HeaderCRUDForm object at
↳0x110289590>,
  'default': <object object at 0x100311200>,
  'here': <plone.app.headeranimation.browser.forms.HeaderCRUDForm object at
↳0x110289590>,
  'loop': {},
  'nothing': None,
  'options': {'args': (<plone.app.headeranimation.browser.forms.
↳AddHeaderAnimationForm object at 0x1102dc490>,)},
  'repeat': <Products.PageTemplates.Expressions.SafeMapping object at 0x110845758>,
  'request': None,
  'root': None,
```

```

    'template': <ImplicitAcquirerWrapper object at 0x11084ff10>,
    'traverse_subpath': [],
    'user': <PropertiedUser 'admin'>,
    'view': <UnauthorizedBinding: context>,
    'views': <zope.app.pagetemplate.viewpagetemplatefile.ViewMapper object at 0x110844310>
  ↪0x110844310>}
Module zope.tales.expressions, line 217, in __call__
Module Products.PageTemplates.Expressions, line 155, in _eval
Module zope.tales.expressions, line 124, in _eval
Module Products.PageTemplates.Expressions, line 105, in trustedBoboAwareZopeTraverse
Module zope.traversing.adapters, line 154, in traversePathElement
- __traceback_info__: (<plone.app.headeranimation.browser.forms.HeaderCRUDForm_
  ↪object at 0x110289590>, '@@standard_macros')
Module zope.traversing.namespace, line 107, in namespaceLookup
TraversalError: (<plone.app.headeranimation.browser.forms.HeaderCRUDForm object at 0x110289590>, '++view++standard_macros')

```

Wrapping is missing from your form object.

Solution:

```

def update(self):
    super(HeaderCRUDForm, self).update()

    addform = self.addform_factory(self, self.request)
    editform = self.editform_factory(self, self.request)

    import zope.interface
    from plone.z3cform.interfaces import IWrappedForm

    zope.interface.alsoProvides(addform, IWrappedForm)
    addform.update()
    editform.update()
    self.subforms = [editform, addform]

```

TraversalError: No traversable adapter found

Traceback:

```

...
* Module ZPublisher.Publish, line 202, in publish_module_standard
* Module ZPublisher.Publish, line 150, in publish
* Module Zope2.App.startup, line 221, in zpublisher_exception_hook
* Module ZPublisher.Publish, line 119, in publish
* Module ZPublisher.mapply, line 88, in mapply
* Module ZPublisher.Publish, line 42, in call_object
* Module Shared.DC.Scripts.Bindings, line 313, in __call__
* Module Shared.DC.Scripts.Bindings, line 350, in _bindAndExec
* Module Products.CMFCore.FSPageTemplate, line 216, in _exec
* Module Products.CMFCore.FSPageTemplate, line 155, in pt_render
* Module Products.PageTemplates.PageTemplate, line 98, in pt_render
* Module zope.pagetemplate.pagetemplate, line 117, in pt_render
Warning: Macro expansion failed
Warning: zope.traversing.interfaces.TraversalError: ('No traversable adapter found',

```

This traceback is followed by long dump of template code internals.

Usual cause: Some add-on product fails to initialize.

Solution: Start Zope in foreground mode (bin/instance fg) to see which product fails.

TypeError: 'ExtensionClass.ExtensionClass' object is not iterable

Traceback:

```
Module ZPublisher.Publish, line 126, in publish
Module ZPublisher.mapply, line 77, in mapply
Module ZPublisher.Publish, line 46, in call_object
Module Shared.DC.Scripts.Bindings, line 322, in __call__
Module Products.PloneHotfix20110531, line 106, in _patched_bindAndExec
Module Shared.DC.Scripts.Bindings, line 359, in _bindAndExec
Module App.special_dtml, line 185, in _exec
Module DocumentTemplate.DT_Let, line 77, in render
Module DocumentTemplate.DT_In, line 647, in renderwob
Module DocumentTemplate.DT_In, line 772, in sort_sequence
Module ZODB.Connection, line 860, in setstate
Module ZODB.Connection, line 914, in _setstate
Module ZODB.serialize, line 612, in setGhostState
Module ZODB.serialize, line 605, in getState
Module zope.interface.declarations, line 756, in Provides
Module zope.interface.declarations, line 659, in __init__
Module zope.interface.declarations, line 45, in __init__
Module zope.interface.declarations, line 1382, in _normalizeargs
Module zope.interface.declarations, line 1381, in _normalizeargs
TypeError: ("'ExtensionClass.ExtensionClass' object is not iterable", <function_
↳Provides at 0x9f04d84>, (<class 'Products.ATContentTypes.content.folder.ATFolder'>,
↳<class 'Products.Carousel.interfaces.ICarouselFolder'>))
```

Condition: This error tends to happen after moving a Data.fs to a new instance that does not have the identical add-ons to the original instance.

In this example traceback the missing add-on is Products.Carousel which provides the marker interface Products.Carousel.interfaces.ICarousel

Solution: Install the missing add-on(s)

TypeError: 'NoneType' object is not callable during upgrade

Traceback:

```
Traceback (innermost last):
  Module ZPublisherEventsBackport.patch, line 77, in publish
  Module ZPublisher.mapply, line 88, in mapply
  Module ZPublisher.Publish, line 42, in call_object
  Module Products.CMFQuickInstallerTool.QuickInstallerTool, line 589, in _
↳installProducts
  Module Products.CMFQuickInstallerTool.QuickInstallerTool, line 526, in _
↳installProduct
    - __traceback_info__: ('mfabrik.plonezohointegration',)
  Module Products.GenericSetup.tool, line 390, in runAllImportStepsFromProfile
    - __traceback_info__: profile-mfabrik.plonezohointegration:default
  Module Products.GenericSetup.tool, line 1179, in _runImportStepsFromContext
  Module Products.GenericSetup.tool, line 1090, in _doRunImportStep
    - __traceback_info__: toolset
```

```
Module Products.GenericSetup.tool, line 128, in importToolset
TypeError: 'NoneType' object is not callable
```

Condition: This error can happen during add-on install run / site upgrade

Reason: This means that your site database contains installed add-on utility objects for which Python code is no longer present.

More pointers for resolving the tool can be found using pdb:

```
(Pdb) tool_id
'portal_newsletters'
```

This happens when you have used Singing and Dancing news letter product. This add-on is problematic and does not uninstall cleanly.

- Reinstall Singing & Dancing
- Uninstall Singing & Dancing
- Hope your site works again

See also:

- *Manually Removing Local Persistent Utilities*
- <http://opensourcehacker.com/2011/06/01/plone-4-upgrade-results-and-steps/>
- <https://pypi.python.org/pypi/wildcard.fixpersistentutilities>

TypeError: argument of type 'NoneType' is not iterable

Traceback:

```
Module ZPublisher.Publish, line 115, in publish
Module ZPublisher.BaseRequest, line 437, in traverse
Module Products.CMFCore.DynamicType, line 147, in __before_publishing_traverse__
Module Products.CMFDynamicViewFTI.fti, line 215, in queryMethodID
Module Products.CMFDynamicViewFTI.fti, line 182, in defaultView
Module Products.CMFFPlone.PloneTool, line 831, in browserDefault
Module plone.app.folder.base, line 65, in index_html
Module plone.folder.ordered, line 202, in __contains__
TypeError: argument of type 'NoneType' is not iterable
```

Reason Plone 3 > Plone 4 migration has not been run. Run the migration in *portal_migrations* under the Management Interface.

TypeError: len() of unsized object in smtplib

Traceback:

```
Traceback (innermost last):
Module ZPublisher.Publish, line 119, in publish
Module ZPublisher.mapply, line 88, in mapply
Module ZPublisher.Publish, line 42, in call_object
Module Products.CMFFormController.FSControllerPageTemplate, line 90, in __call__
Module Products.CMFFormController.BaseControllerPageTemplate, line 28, in _call
Module Products.CMFFormController.ControllerBase, line 231, in getNext
Module Products.CMFFormController.Actions.TraverseTo, line 38, in __call__
```

```

Module ZPublisher.mapply, line 88, in mapply
Module ZPublisher.Publish, line 42, in call_object
Module Products.CMFFormController.FSControllerPythonScript, line 104, in __call__
Module Products.CMFFormController.Script, line 145, in __call__
Module Products.CMFCore.FSPythonScript, line 140, in __call__
Module Shared.DC.Scripts.Bindings, line 313, in __call__
Module Shared.DC.Scripts.Bindings, line 350, in _bindAndExec
Module Products.CMFCore.FSPythonScript, line 196, in _exec
Module None, line 102, in order_email
- <FSControllerPythonScript at /MySite/order_email>
- Line 102
Module Products.SecureMailHost.SecureMailHost, line 246, in secureSend
Module Products.SecureMailHost.SecureMailHost, line 276, in _send
Module Products.SecureMailHost.mail, line 126, in send
Module smtplib, line 576, in login
Module smtplib, line 536, in encode_cram_md5
Module hmac, line 50, in __init__
TypeError: len() of unsized object

```

Reason: Your SMTP password has been set empty. Please reset your SMTP password in *Mail* control panel.

See also:

<http://plone.293351.n2.nabble.com/Plone-3-3-5-sending-emails-len-of-unsized-object-error-NO-ESMTP-PASSWORD-tp5415484p54.html>

Unauthorized: The object is marked as private

Traceback:

```

File "/home/moo/twinapex/parts/zope2/lib/python/zope/tales/expressions.py", line 124, in
↳ in _eval
  ob = self._traverser(ob, element, econtext)
File "/home/moo/twinapex/parts/zope2/lib/python/Products/PageTemplates/Expressions.py
↳ ", line 105, in trustedBoboAwareZopeTraverse
  request=request)
File "/home/moo/twinapex/parts/zope2/lib/python/zope/traversing/adapters.py", line
↳ 164, in traversePathElement
  return traversable.traverse(nm, further_path)
File "/home/moo/twinapex/parts/zope2/lib/python/zope/traversing/adapters.py", line 44,
↳ in traverse
  attr = getattr(subject, name, _marker)
File "/home/moo/twinapex/parts/zope2/lib/python/Shared/DC/Scripts/Bindings.py", line
↳ 184, in __getattr__
  return guarded_getattr(self._wrapped, name, default)
File "/home/moo/twinapex/parts/zope2/lib/python/AccessControl/ImplPython.py", line
↳ 563, in validate
  self._context)
File "/home/moo/twinapex/parts/zope2/lib/python/AccessControl/ImplPython.py", line
↳ 443, in validate
  accessed, container, name, value, context)
File "/home/moo/twinapex/parts/zope2/lib/python/AccessControl/ImplPython.py", line
↳ 808, in raiseVerbose
  raise Unauthorized(text)
Unauthorized: The object is marked as private. Access to 'showVideo' of (Products.
↳ Five.metaclass.SimpleViewClass from /home/moo/twinapex/src/mfabrik.app/mfabrik/app/
↳ browser/campaigntopview.pt object at 0x11003a0c) denied.

```


Condition: This error is raised when you try to access view functions or objects for a view, which you call manually from the code.

Reason: View acquisition chain is not properly set up and the security manager cannot traverse acquisition chain parents to properly determine permissions.

Solution: You need to use `__of__()` method to set-up the acquisition chain for the view:

```
def getHeadingView(self):
    """
    Check if we have campaign view available for this content and use it.
    """
    view = queryMultiAdapter((self.context, self.request), name="mfabrik_heading")
    view = view.__of__(self.context) # <----- here
    return view
```

Unknown message (kss optimized for production mode) in JavaScript console

This is a KSS error message. KSS is a technology used in Plone 3 and started to be phased out in Plone 4.

Possible causes:

- Problems with KSS files (see `portal_kss` registry)
- Browser bugs (Google around for the fixes)

Solution:

- Go to `portal_kss`
- Remove are stale entries (missing files, marked on red)

Also:

- Put `portal_kss` for debug mode (in development environment)

ValueError: Non-zero version length. Versions aren't supported.

Traceback:

```
File "/Users/moo/code/buildout-cache/eggs/zope.component-3.7.1-py2.6.egg/zope/
↳ component/registry.py", line 323, in subscribers
    return self.adapters.subscribers(objects, provided)
File "/Users/moo/code/buildout-cache/eggs/ZODB3-3.9.5-py2.6-macosx-10.6-i386.egg/ZODB/
↳ Connection.py", line 838, in setstate
    self._setstate(obj)
File "/Users/moo/code/buildout-cache/eggs/ZODB3-3.9.5-py2.6-macosx-10.6-i386.egg/ZODB/
↳ Connection.py", line 888, in _setstate
    p, serial = self._storage.load(obj._p_oid, '')
File "/Users/moo/code/buildout-cache/eggs/ZODB3-3.9.5-py2.6-macosx-10.6-i386.egg/ZEO/
↳ ClientStorage.py", line 810, in load
    data, tid = self._server.loadEx(oid)
File "/Users/moo/code/buildout-cache/eggs/ZODB3-3.9.5-py2.6-macosx-10.6-i386.egg/ZEO/
↳ ServerStub.py", line 176, in loadEx
    return self.rpc.call("loadEx", oid)
File "/Users/moo/code/buildout-cache/eggs/ZODB3-3.9.5-py2.6-macosx-10.6-i386.egg/ZEO/
↳ zrpc/connection.py", line 703, in call
    raise inst # error raised by server
ValueError: Non-zero version length. Versions aren't supported.
```

Condition: When trying to open any page

Reason: Most likely a corrupted Data.fs. Stop zeoserver. Recopy Data.fs. Recopy blobs.

See also:

- <http://stackoverflow.com/questions/8387902/plone-upgrade-3-3-5-to-plone-4-1-2>
- <https://mail.zope.org/pipermail/zodb-dev/2010-September/013620.html>

Zope suddenly dies on OSX without a reason

Symptoms: you do a HTTP request to a Plone site running OSX. Zope quits without a reason.

Reason: Infinite recursion is not properly handled by Python on OSX. This is because OSX C stack size is smaller than Python default stack size. The underlying Python interpreter dies before being able to raise stack size limit exception.

Solution

Edit `python-2.4/lib/python2.4/site.py` or corresponding Python interpreter `site.py` file (Python site installation customization file).

Put in to the first code line:

```
sys.setrecursionlimit(800)
```

This will force smaller Python stack not exceeding native OSX C stack. You might want to test other values and report back the findings.

See also:

<http://blog.crowproductions.de/2008/12/14/a-buildout-to-tame-the-snake-pit/> (comments)

from zopeskel.basic_namespace import BasicNamespace

When starting ZopeSkel:

```
File "/home/moo/code/python2/parts/opt/lib/python2.6/pkgutil.py", line 238, in load_
↳ module
  mod = imp.load_module(fullname, self.file, self.filename, self.etc)
File "/home/moo/code/plonecommunity/eggs/ZopeSkel-2.19-py2.6.egg/zopeskel/__init__.py
↳ ", line 2, in <module>
  from zopeskel.basic_namespace import BasicNamespace
```

Or on paster with local commands:

```
File "/fast/buildout-cache/eggs/templer.core-1.0b4-py2.6.egg/templer/core/basic_
↳ namespace.py", line 3, in <module>
  from templer.core.base import BaseTemplate
File "/fast/buildout-cache/eggs/templer.core-1.0b4-py2.6.egg/templer/core/base.py",
↳ line 8, in <module>
  from paste.script import command
ImportError: cannot import name command
```

System-wide templer / paster / zopeskel installation is affecting your buildout installation.

Remove system-wide installation:

```
rm -rf /home/moo/code/python2/python-2.6/lib/python2.6/site-packages/ZopeSkel-2.19-
↳ py2.6.egg/
```

Re-run buildout.

Enjoy.

getUtility() fails: ComponentLookupError

Traceback:

```
-> filter = getUtility(IConvergenceMediaFilter)
(Pdb) n
ComponentLookupError: <zope.component.interfaces.ComponentLookupError instance at_
↳0x1038166c>
```

Solution: Make sure that your class object implements in the utility interface in the question:

```
class ConvergedMediaFilter(object):
    zope.interface.implements(IConvergenceMediaFilter)
```

get_language: 'NoneType' object has no attribute 'getLocaleID'

Traceback:

```
Module ZPublisher.Publish, line 202, in publish_module_standard
Module ZPublisherEventsBackport.patch, line 115, in publish
Module plone.app.linkintegrity.monkey, line 21, in zpublisher_exception_hook_wrapper
Module Zope2.App.startup, line 221, in zpublisher_exception_hook
Module ZPublisherEventsBackport.patch, line 77, in publish
Module ZPublisher.mapply, line 88, in mapply
Module ZPublisher.Publish, line 42, in call_object
Module Products.Five.browser.metaconfigure, line 417, in __call__
Module Shared.DC.Scripts.Bindings, line 313, in __call__
Module Shared.DC.Scripts.Bindings, line 350, in _bindAndExec
Module Products.PageTemplates.PageTemplateFile, line 129, in _exec
Module Products.CacheSetup.patch_cmf, line 126, in PT_pt_render
Warning: Macro expansion failed
Warning: exceptions.TypeError: ('Could not adapt', <HTTPRequest, URL=http://mansikki.
↳redinnovation.com:9666/isleofback/sisalto/etusivu/isleofbackfrontpage_view>,
↳<InterfaceClass zope.interface.interfaces.IUserPreferredLanguages>)
Module zope.tal.talinterpreter, line 271, in __call__
Module zope.tal.talinterpreter, line 346, in interpret
Module zope.tal.talinterpreter, line 891, in do_useMacro
Module zope.tal.talinterpreter, line 346, in interpret
Module zope.tal.talinterpreter, line 536, in do_optTag_tal
Module zope.tal.talinterpreter, line 521, in do_optTag
Module zope.tal.talinterpreter, line 516, in no_tag
Module zope.tal.talinterpreter, line 346, in interpret
Module zope.tal.talinterpreter, line 534, in do_optTag_tal
Module zope.tal.talinterpreter, line 516, in no_tag
Module zope.tal.talinterpreter, line 346, in interpret
Module zope.tal.talinterpreter, line 745, in do_insertStructure_tal
Module Products.PageTemplates.Expressions, line 223, in evaluateStructure
Module zope.tales.tales, line 696, in evaluate
URL: file:/srv/plone/saariselka.fi/src/plonetheme.isleofback/plonetheme/isleofback/
↳skins/plonetheme_isleofback_custom_templates/main_template.pt
Line 58, Column 4
Expression: <StringExpr u'plone.htmlhead.links'>
```

Names:

```
{'container': <IsleofbackFrontpage at /isleofback/sisalto/etusivu>,
'context': <IsleofbackFrontpage at /isleofback/sisalto/etusivu>,
'default': <object object at 0x7fd445785220>,
'here': <IsleofbackFrontpage at /isleofback/sisalto/etusivu>,
'loop': {},
'nothing': None,
'options': {'args': (<Products.Five.metaclass.SimpleViewClass from /srv/plone/
↳saariselka.fi/src/isleofback.app/isleofback/app/browser/isleofbacknewfrontpageview.
↳pt object at 0xbaa9910>,)},
'repeat': <Products.PageTemplates.Expressions.SafeMapping object at 0xcd1b3f8>,
'request': <HTTPRequest, URL=http://mansikki.redinnovation.com:9666/isleofback/
↳sisalto/etusivu/isleofbackfrontpage_view>,
'root': <Application at >,
'template': <ImplicitAcquirerWrapper object at 0xcd208d0>,
'traverse_subpath': [],
'user': <SpecialUser 'Anonymous User'>,
'view': <Products.Five.metaclass.SimpleViewClass from /srv/plone/saariselka.fi/src/
↳isleofback.app/isleofback/app/browser/isleofbacknewfrontpageview.pt object at
↳0xbaa9910>,
'views': <zope.app.pagetemplate.viewpagetemplatefile.ViewMapper object at 0xcd20d90>}
```

```
Module Products.Five.browser.providerexpression, line 37, in __call__
Module plone.app.viewletmanager.manager, line 83, in render
Module plone.memoize.volatile, line 265, in replacement
Module plone.app.layout.links.viewlets, line 28, in render_cachekey
Module plone.app.layout.links.viewlets, line 19, in get_language
```

```
AttributeError: <exceptions.AttributeError instance at 0xcd1bb48> (Also, the
↳following error occurred while attempting to render the standard error message,
↳please see the event log for full details: 'NoneType' object has no attribute
↳'getLocaleID')
```

Some sort of Products.CacheSetup related problem on Plone 3.3.x, hiding the real error. Zope component architecture loading has failed (you are missing critical bits). This is just the first entry where it tries to use an unloaded code.

Start your instance on the foreground and you should see the actual error.

importToolset: TypeError: 'NoneType' object is not callable

Traceback:

```
Module ZPublisher.Publish, line 47, in call_object
Module Products.CMFQuickInstallerTool.QuickInstallerTool, line 575, in installProducts
Module Products.CMFQuickInstallerTool.QuickInstallerTool, line 512, in installProduct
- __traceback_info__: ('plone.app.registry',)
Module Products.GenericSetup.tool, line 323, in runAllImportStepsFromProfile
- __traceback_info__: profile-plone.app.registry:default
Module Products.GenericSetup.tool, line 1080, in _runImportStepsFromContext
Module Products.GenericSetup.tool, line 994, in _doRunImportStep
- __traceback_info__: toolset
Module Products.GenericSetup.tool, line 123, in importToolset
```

Condition: This happens when you try to install an add-on product through Add-ons configuration panel.

Reason: You have leftovers from some old add-on installation (persistent tool) and Python egg code is no longer present for this tool.

You should see a warning in logs giving you a hint when running add-on installer:

```
2011-05-29 16:40:25 INFO GenericSetup.toolset Class Products.Notifica.NotificaTool.
↳NotificaTool not found for tool notifica_tool
```

Solution: see information below (Removing portal tools part)

- *Manually Removing Local Persistent Utilities*

Example: start site debug shell:

```
bin/instance debug
```

Then run the script for your site id and problem tool id:

```
bad_tool = 'notifica_tool'
site = app.yoursiteid

setup_tool = site.portal_setup
toolset = setup_tool.getToolsetRegistry()
if bad_tool in toolset._required.keys():
    del toolset._required[bad_tool]
    setup_tool._toolset_registry = toolset
else:
    print "Tool not found:" + bad_tool

import transaction ; transaction.commit()
app._p_jar.sync()
```

In debug shell you can also check what all leftovers toolset contains:

```
>>> toolset._required.keys()
['portal_historyidhandler', 'portal_actions', 'portal_skins', 'portal_form_controller
↳',
'portal_workflow', 'portal_catalog', 'portal_languages', 'kupu_library_tool', 'portal_
↳diff',
'portal_repository', 'reference_catalog', 'portal_groupdata', 'portal_search_and_
↳replace',
'portal_atct', 'mimetypes_registry', 'portal_purgepolicy', 'formgen_tool', 'uid_
↳catalog',
'error_log', 'portal_modifier', 'portal_discussion', 'portal_actionicons', 'portal_
↳calendar', 'portal_metadata', 'portal_url',
'portal_archivist', 'portal_tinymce', 'portal_factory', 'content_type_registry',
↳'portal_groups', 'portal_controlpanel',
'portal_uidannotation', 'portal_transforms', 'portal_memberdata', 'portal_javascripts
↳', 'portal_registration', 'portal_css',
'portal_facets_catalog', 'portal_password_reset', 'plone_utils', 'caching_policy_
↳manager',
'portal_historiesstorage', 'portal_undo', 'portal_placeful_workflow', 'translation_
↳service',
'archetype_tool', 'portal_view_customizations', 'portal_syndication', 'portal_
↳quickinstaller', 'portal_uidhandler',
'portal_referencefactories', 'portal_interface', 'portal_facetednavigation', 'portal_
↳membership',
'MailHost', 'portal_properties', 'portal_migration', 'portal_types', 'portal_
↳uidgenerator']
```

See also:

<http://plone.293351.n2.nabble.com/importToolset-NoneType-object-is-not-callable-upon-product-install-td5553065.html>

z3c.form based form updateWidgets() raises ComponentLookupError

Case: missing plone.app.z3cform migration

Traceback:

```
Traceback (innermost last):
  Module ZPublisher.Publish, line 126, in publish
  Module ZPublisher.mapply, line 77, in mapply
  Module ZPublisher.Publish, line 46, in call_object
  Module z3c.form.form, line 215, in __call__
  Module z3c.form.form, line 208, in update
  Module plone.z3cform.patch, line 21, in BaseForm_update
  Module z3c.form.form, line 149, in update
  Module z3c.form.form, line 129, in updateWidgets
  Module zope.component._api, line 109, in getMultiAdapter
ComponentLookupError: ((<Products.Five.metaclass.EditForm object at 0x117a97dd0>,
↳<HTTPRequest, URL=http://localhost:8080/folder_xxx/xxxngta/@@dgftreeselect-test>,
↳<PloneSite at /folder_xxx/xxxngta>), <InterfaceClass z3c.form.interfaces.IWidgets>,
↳u'')
```

Reason: You are running Plone 4 with `plone.app.directives` form which does not open. The reason is that you most likely have old `plone.app.z3cform` installation which is not upgraded properly. In particular, the following layer is missing

```
<layer name="plone.app.z3cform" interface="plone.app.z3cform.interfaces.
↳IPloneFormLayer" />
```

This enables `z3c.form` widgets on a Plone site.

Solution: `portal_setup > Import`. Choose profile *Plone z3cform support*. and import. The layer gets properly inserted to your site database.

Buildout troubleshooting

Description

How to solve problems related to running buildout and some common exceptions you might encounter when running buildout for Plone.

Introduction

This document tells how to resolve buildout problems.

Network errors and timeouts

The usual reason for download error or timeout is that either

- `pypi.python.org` server is down, or

- one of `plone.org` servers is down, or
- other Python package source server is down.

Here are instructions how to deal with community servers down situations

- <http://jacobian.org/writing/when-pypi-goes-down/>

Mirrors

- <http://www.pypi-mirrors.org/>

Individual package failing outside PyPI

To figure out which file buildout tries to download, usually the only way is to use `buildout -D pdb` debug mode and step up in stack frames to see what is going on.

parts/instance/etc/zope.conf: [Errno 2] No such file or directory

You see this error when trying to start Plone. This means that buildout did not complete correctly and did not generate configuration files.

Rerun buildout and fix errors in `buildout.cfg` based on buildout command output.

Buildout and SyntaxErrors

You may see `SyntaxError` exceptions when running buildout:

```
SyntaxError: ('return' outside function", ('/usr/local/Plone/buildout-cache/eggs/
↳tmpzTKrEI/Products.ATExtensions-1.1a3-py2.6.egg/Products/ATExtensions/skins/at_
↳extensions/getDisplayView.py', 11, None, 'return value\n'))
```

They are harmless.

The reason: Buildout uses a Python tool called `setuptools` internally to install the packages. `Setuptools` scans all `.py` files inside the Python package and assumes they are Python modules. However, Plone has something called *RestrictedPython*. `RestrictedPython` allows untrusted users to execute Python code in Plone (Python Scripts in the Management Interface). `RestrictedPython` scripts use slightly modified Python syntax compared to plain Python modules.

`Setuptools` does not know which files are normal `.py` and which files are `RestrictedPython` and tries to interpret them all using standard Python syntax rules. Then it fails. However, `setuptools` only tries to scan files (in order to see if they are zip-safe) but still installs them correctly. No harm done.

Version conflicts

Buildout gives you an error if there is a dependency shared by two components, and one of the components wants to have a different version of this dependency.

Example:

```
Installing.
Getting section zeoserver.
Initializing part zeoserver.
Error: There is a version conflict.
```

```
We already have: zope.component 3.8.0
but five.localsitemanager 1.1 requires 'zope.component<3.6dev'.
```

If your buildout is fetching strange versions:

- try running buildout in verbose mode: `bin/buildout -vvv`
- Use `show-picked-versions` (see below)
- Manually pin down version in the `[versions]` section of your buildout.

Further reading:

- <http://maurits.vanrees.org/weblog/archive/2010/08/fake-version-pinning>
- <http://www.uwosh.edu/ploneprojects/documentation/how-tos/how-to-use-buildout-to-pin-product-versions>

Show picked versions

In order to show which versions were picked by buildout - or in other words, versions were not pinned anywhere - use this feature.

Buildout will show automatically picked Python egg versions at the end of the output. The output may be copy pasted in your versions section.

Add to your `buildout.cfg`:

```
[buildout]
show-picked-versions
```

Extracting version numbers from instance script

Example:

```
cat bin/instance | grep eggs | sed -r 's#.*eggs/(.*)-py2.[0-9].*#\1#g' | sed -r 's#-#<br>↪= #g' | sed -r 's#_#-#g' | grep -E ' = [0-9\.]' | xargs -0 echo -e "[versions]\n" |<br>↪sed -r 's#^\s+##g' > versions-extracted.cfg; cat versions-extracted.cfg
```

More info

- <http://davidjb.com/blog/2011/06/extracting-a-buildout-versions-cfg-from-a-zope-instance-script/>

Plone 3.1

Plone 3.1 and earlier are not eggified. Below are links how to keep Plone 3.1 and earlier buildouts running.

See:

- <http://www.netsight.co.uk/blog/resurrecting-old-plone-3-buildouts>

Getting distribution for distribute

You try to run buildout, but it is stuck in a loop:


```

Getting distribution for 'distribute'.
Getting distribution for 'distribute'.
....
Getting distribution for 'distribute'.
Getting distribution for 'distribute'.
Getting distribution for 'distribute'.

```

Your system-wide Distribute version is older than the latest release. Buildout tries to update it, but since system wide site-packages version overrides anything buildout can do, it is stuck in a loop.

Fix: update Distribute in system-wide Python:

```

easy_install -U Distribute
Searching for Distribute
Reading https://pypi.python.org/simple/Distribute/
Reading http://packages.python.org/distribute
Best match: distribute 0.6.12
Downloading https://pypi.python.org/packages/source/d/distribute/distribute-0.6.12.
↳tar.gz#md5=5a52e961f8d8799d243fe8220f9d760e
Processing distribute-0.6.12.tar.gz
Running distribute-0.6.12/setup.py -q bdist_egg --dist-dir /tmp/easy_install-jlL3e7/
↳distribute-0.6.12/egg-dist-tmp-IV9SiQ
Before install bootstrap.
Scanning installed packages
Setuptools installation detected at /home/moo/py24/lib/python2.4/site-packages
Non-egg installation
Removing elements out of the way...
Already patched.
/home/moo/py24/lib/python2.4/site-packages/setuptools-0.6c11-py2.4.egg-info already_
↳patched.
After install bootstrap.
/home/moo/py24/lib/python2.4/site-packages/setuptools-0.6c11-py2.4.egg-info already_
↳exists
Removing distribute 0.6.10 from easy-install.pth file
Adding distribute 0.6.12 to easy-install.pth file
Installing easy_install script to /home/moo/py24/bin
Installing easy_install-2.4 script to /home/moo/py24/bin

```

UnknownExtra: zope.i18n 0.0 has no such extra feature 'zcml'

You get the following traceback when running buildout:

```

File "/home/moo/rtv/eggs/plone.recipe.zope2instance-2.7-py2.4.egg/plone/recipe/
↳zope2instance/__init__.py", line 93, in update
    requirements, ws = self.egg.working_set()
File "/home/moo/rtv/eggs/zc.recipe.egg-1.1.0-py2.4.egg/zc/recipe/egg/egg.py", line_
↳93, in working_set
    allow_hosts=self.allow_hosts,
File "/tmp/tmpGFbvPP/zc.buildout-1.5.0b2-py2.4.egg/zc/buildout/easy_install.py", _
↳line 800, in install
File "/tmp/tmpGFbvPP/zc.buildout-1.5.0b2-py2.4.egg/zc/buildout/easy_install.py", _
↳line 660, in install
File "/home/moo/py24/lib/python2.4/site-packages/distribute-0.6.10-py2.4.egg/pkg_
↳resources.py", line 551, in resolve
    requirements.extend(dist.requires(req.extras)[::-1])
File "/home/moo/py24/lib/python2.4/site-packages/distribute-0.6.10-py2.4.egg/pkg_
↳resources.py", line 2164, in requires

```

```
raise UnknownExtra(
UnknownExtra: zope.i18n 0.0 has no such extra feature 'zcml'
```

You might be using an add-on meant for Plone 4 with Plone 3. Check if `setup.py` contains *Zope2* as a dependency. If it does, then you need to use earlier version of the add-on for your Plone 3 site.

More info:

- http://groups.google.com/group/singing-dancing/browse_thread/thread/331cdf78cf371ed

We already have: zope.interface 4.0.3

Example:

```
Getting distribution for 'zope.testing==3.9.7'.
warning: no files found matching 'sampletests' under directory 'src'
Got zope.testing 3.9.7.
While:
  Installing.
  Getting section test.
  Initializing section test.
  Installing recipe zc.recipe.testrunner.
Error: There is a version conflict.
We already have: zope.interface 4.0.3
```

Your system Python or virtualenv'd Python already has `zope.interface` library installed. A lot of Python software uses this library. However, the system version is wrong and cannot be overridden.

Solutions.

For virtualenv: `rm -rf ~/code/plone-venv/lib/python2.7/site-packages/zope.interface-4.0.3-py2.7-macosx-10.8-x86_64.egg`

For system Python: You need to create a virtualenv'd Python and to use it to drive buildout, so that there is no conflict with `zope.interface` versions.

We already have: zope.location 3.4.0

When running buildout, Plone 3.3.5:

```
While:
  Installing.
  Getting section zoepgy.
  Initializing section zoepgy.
  Getting option zoepgy:eggs.
  Getting section client1.
  Initializing section client1.
  Getting option client1:zeo-address.
  Getting section zeo.
  Initializing part zeo.
Error: There is a version conflict.
We already have: zope.location 3.4.0
but zope.traversing 3.13 requires 'zope.location>=3.7.0'.
```

Solution:

```
rm -rf fake-eggs/*
bin/buildout install zope2
bin/buildout
```

ImportError: No module named lxml

lxml as a PyPi package dependency fails even though it is clearly installed.

Example traceback when running buildout:

```
...
Processing openxmllib-1.0.6.tar.gz
<snip Unpacking... >
Running openxmllib-1.0.6/setup.py bdist_egg --dist-dir /tmp/easy_install-Urh6x4/
→openxmllib-1.0.6/egg-dist-tmp-ju0TuT
Traceback (most recent call last):
<snip Traceback... >
  File "setup.py", line 5, in <module>
    File "/tmp/easy_install-Urh6x4/openxmllib-1.0.6/openxmllib/__init__.py", line 17,
→in <module>
    File "/tmp/easy_install-Urh6x4/openxmllib-1.0.6/openxmllib/wordprocessing.py", line
→5, in <module>
    File "/tmp/easy_install-Urh6x4/openxmllib-1.0.6/openxmllib/document.py", line 14,
→in <module>
ImportError: No module named lxml
An error occurred when trying to install openxmllib 1.0.6. Look above this message
→for any errors that were output by easy_install.
While:
  Installing plone-core-addons.
  Getting distribution for 'openxmllib>=1.0.6'.
Error: Couldn't install: openxmllib 1.0.6
```

Solution: ensure lxml compilation happens before openxmllib is being compiled.

For instance, if you are installing something like `Products.OpenXml`, you will have likely included this egg under your Plone [instance] section of your buildout. You should consider using something like `collective.recipe.staticlxml` to build lxml and to do this *before* this egg's installation is invoked. Like so in your `buildout.cfg`:

```
[buildout]
parts =
    lxml
    ...
    instance
...

[lxml]
recipe = z3c.recipe.staticlxml
egg = lxml
```

More information:

- <http://www.niteoweb.com/blog/order-of-parts-when-compiling-lxml>
- <http://plone.293351.n2.nabble.com/lxml-installs-but-Products-OpenXml-openxmllib-can-t-see-it-t5565184p5565184.html>

Can't run bootstrap.py - VersionConflict for zc.buildout

Traceback when running python bootstrap.py:

```
Traceback (most recent call last):
  File "/Users/moo/code/collective.buildout.python/parts/opt/lib/python2.6/pdb.py",
↳line 1283, in main
    pdb._runscript(mainpyfile)
  File "/Users/moo/code/collective.buildout.python/parts/opt/lib/python2.6/pdb.py",
↳line 1202, in _runscript
    self.run(statement)
  File "/Users/moo/code/collective.buildout.python/parts/opt/lib/python2.6/bdb.py",
↳line 368, in run
    exec cmd in globals, locals
  File "<string>", line 1, in <module>
  File "bootstrap.py", line 256, in <module>
    ws.require(requirement)
  File "/Users/moo/code/collective.buildout.python/python-2.6/lib/python2.6/site-
↳packages/distribute-0.6.8-py2.6.egg/pkg_resources.py", line 633, in require
    needed = self.resolve(parse_requirements(requirements))
  File "/Users/moo/code/collective.buildout.python/python-2.6/lib/python2.6/site-
↳packages/distribute-0.6.8-py2.6.egg/pkg_resources.py", line 535, in resolve
    raise VersionConflict(dist, req) # XXX put more info here
VersionConflict: (zc.buildout 1.5.0b2 (/Users/moo/code/collective.buildout.python/
↳python-2.6/lib/python2.6/site-packages/zc.buildout-1.5.0b2-py2.6.egg), Requirement.
↳parse('zc.buildout==1.5.2'))
```

Solution: update the zc.buildout installed in your system Python:

```
easy_install -U zc.buildout
```

An error occurred when trying to install lxml - error: Setup script exited with error: command 'gcc' failed with exit status 1

Traceback when running buildout:

```
...
src/lxml/lxml.etree.c:143652: error: '__pyx_v_4lxml_5etree_XSLT_DOC_DEFAULT_LOADER'
↳undeclared (first use in this function)
src/lxml/lxml.etree.c:143652: error: 'xsltDocDefaultLoader' undeclared (first use in
↳this function)
src/lxml/lxml.etree.c:143661: error: '__pyx_f_4lxml_5etree__xslt_doc_loader'
↳undeclared (first use in this function)
error: Setup script exited with error: command 'gcc' failed with exit status 1
An error occurred when trying to install lxml 2.2.8. Look above this message for any
↳errors that were output by easy_install.
While:
  Installing instance.
  Getting distribution for 'lxml==2.2.8'.
Error: Couldn't install: lxml 2.2.8
```

Solution: install the libxml and libxslt development headers.

On Ubuntu/Debian you could do this as follows:

```
sudo apt-get install libxml2-dev libxslt-dev
```

VersionConflict: distribute 0.6.19

When running buildout you see something like this:

```

File "/home/danieltordable.es/buildout-cache/eggs/zc.buildout-1.4.4-py2.6.egg/zc/
↪buildout/easy_install.py", line 606, in _maybe_add_setuptools
    if ws.find(requirement) is None:
File "/home/danieltordable.es/buildout-cache/eggs/distribute-0.6.19-py2.6.egg/pkg_
↪resources.py", line 474, in find
    raise VersionConflict(dist, req)          # XXX add more info
VersionConflict: (distribute 0.6.19 (/home/danieltordable.es/buildout-cache/eggs/
↪distribute-0.6.19-py2.6.egg), Requirement.parse('distribute==0.6.15'))

```

Buildout uses the system-wide Distribute installation (python-distribute or similar package, depends on your OS). To fix this, you need to update system-wide distribution.

Note: It is preferred to do your Python + buildout installation in a *virtualenv*, in order not to break your OS

Update Distribute (Plone universal installer, using supplied easy_install script):

```
python/bin/easy_install -U Distribute
```

Update Distribute (OSX/Ubuntu/Linux):

```
easy_install -U Distribute
```

argparse 1.2.1

If you get:

```

While:
  Installing.
  Loading extensions.
Error: There is a version conflict.
We already have: argparse 1.2.1

```

Rerun bootstrap.py with the correct Python interpreter.

Error: Picked: <some.package> = <some.version>

If you get something like this:

```

We have the distribution that satisfies 'zc.recipe.testrunner==1.2.1'.
Installing 'collective.recipe.backup'.
Picked: collective.recipe.backup = 2.4
Could't load zc.buildout entry point default
from collective.recipe.backup:
Picked: collective.recipe.backup = 2.4.
While:
  Installing.
  Getting section backup.
  Initializing section backup.
  Installing recipe collective.recipe.backup.

```

```
Getting distribution for 'collective.recipe.backup'.
Error: Picked: collective.recipe.backup = 2.4
```

This means that your buildout has “allow picked versions” set to false. You need to pin the version for the picked version (or turn on “allow picked versions”).

Buildout error: Not a recognized archive type

If you run across an error like this when running buildout:

```
...
Installing instance.
Getting distribution for 'collective.spaces'.
error: Not a recognized archive type: /home/plone/.buildout/downloads/dist/collective.
↳spaces-1.0.zip
```

the error is likely stemming from an incorrect download of this egg. Check the given file to ensure that the file is correct (for instance, it is a non-zero length file or verifying the content using something like `md5sum`) before delving deep into your Python install’s workings. This error makes it look as if your Python install doesn’t have support for this type of archive, but in fact it can be caused by a corrupt download.

Distribute / setuptools tries to mess with system Python and Permission denied

When running `bootstrap.py` your buildout files because it tries to write to system-wide Python installation.

Example:

```
Getting distribution for 'distribute==0.6.24'.
Before install bootstrap.
Scanning installed packages
No setuptools distribution found
warning: no files found matching 'Makefile' under directory 'docs'
warning: no files found matching 'indexsidebar.html' under directory 'docs'
After install bootstrap.
Creating /srv/plone/python/python-2.7/lib/python2.7/site-packages/setuptools-0.6c11-
↳py2.7.egg-info
error: /srv/plone/python/python-2.7/lib/python2.7/site-packages/setuptools-0.6c11-py2.
↳7.egg-info: Permission denied
An error occurred when trying to install distribute 0.6.24. Look above this message_
↳for any errors that were output by easy_install.
While:
  Bootstrapping.
  Getting distribution for 'distribute==0.6.24'.
Error: Couldn't install: distribute 0.6.24
```

Solution:

This bug has been fixed in Distribute 0.6.27 - make sure your system-wide Python uses this version or above:

```
sudo /srv/plone/python/python-2.7/bin/easy_install -U Distribute
```

UnboundLocalError: local variable ‘clients’ referenced before assignment

Example traceback when running buildout:

```

Traceback (most recent call last):
  File "/srv/plone/x/eggs/zc.buildout-1.4.4-py2.7.egg/zc/buildout/buildout.py", line
↪1683, in main
    getattr(buildout, command)(args)
  File "/srv/plone/x/eggs/zc.buildout-1.4.4-py2.7.egg/zc/buildout/buildout.py", line
↪555, in install
    installed_files = self[part]._call(recipe.install)
  File "/srv/plone/x/eggs/zc.buildout-1.4.4-py2.7.egg/zc/buildout/buildout.py", line
↪1227, in _call
    return f()
  File "/srv/plone/x/eggs/plone.recipe.unifiedinstaller-4.3.1-py2.7.egg/plone/recipe/
↪unifiedinstaller/__init__.py", line 65, in install
    for part in clients
UnboundLocalError: local variable 'clients' referenced before assignment

```

Solution: Your buildout contains leftovers from the past. Remove `clients` variable in `[unifiedinstaller]` section.

error: None

This means `.tar.gz` is corrupted:

```

error: None
An error occurred when trying to install lxml 2.3.6. Look above this message for any
↪errors that were output by easy_install.
While:
  Installing instance.
  Getting distribution for 'lxml==2.3.6'.
Error: Couldn't install: lxml 2.3.6

```

Buildout download cache is corrupted. Run `bin/buildout -vvv` for more info. Then do something like this:

```

# Corrupted .tar.gz download
rm /Users/mikko/code/buildout-cache/downloads/dist/lxml-2.3.6.tar.gz

```

Mac OS X Error: Couldn't install: lxml 3.4.4

If you got the error `Couldn't install: lxml 3.4.4` while using the Plone 5.0 unified installer on Mac OS X 10.11.1 El Capitan, you should update your Xcode command line tools:

```
xcode-select --install
```

then re-run `bin/buildout`.

Unicode encoding and decoding

Introduction: Why unicode is difficult?

Python 2.x does not make a clear distinction between:

- 8-bit strings (byte data)
- 16-bit unicode strings (character data)

Developers use these two formats interchangeably, because it is so easy and Python does not warn you about this.

However, it will only work as long as the input does not encounter any international, non-ASCII, characters. When 8-bit encoded string data and 16-bit raw Unicode string data gets mixed up, by being run through encoding first, really nasty things start to happen.

Read more:

- <http://evanjones.ca/python-utf8.html>

safe_unicode()

Plone's core contains a helper function which allows you to safely decode strings to unicode without fear of UnicodeDecodeException. Use this in your own code to decode unicode in the cases you are not sure if the input is 8-bit bytestrings or real unicode strings.

<https://github.com/plone/Products.CMFPlone/blob/master/Products/CMFPlone/utils.py#L434>

Example:

```
# -*- coding: utf-8 -*-

from Products.CMFPlone.utils import safe_unicode

foobar = safe_unicode("Ärrinmurrin taas on Plonea joku jättänyt dokumentoimatta")
```

sys.setdefaultencoding()

Python has a **system-wide** setting to enforce encoding of all unicode input automatically to utf-8 when used as 8-bit string.

Warning: This is a wrong way to fix things and it will break other things. You have been warned.

- <http://tarekziade.wordpress.com/2008/01/08/syssetdefaultencoding-is-evil/>

There is also `sitecustomization.py` trick to set `sys.setdefaultencoding("utf-8")` on per-script basis

- <http://stackoverflow.com/a/7892892/315168>

UnicodeEncodeError

UnicodeEncodeError: 'ascii' codec can't encode character u'xe4' in position 4: ordinal not in range(128)

This is usually because you are trying to output/store unicode data using outdated methods, e.g.

- printing,
- logging,
- using 7-bit ids ...

Instead of:

```
print foo
```


do:

```
print foo.encode("utf-8") # You are sure this is a unicode string
```

Filtering example:

```
def safe_print(x):
    """ Do not die on bad input when doing debug prints """
    if type(x) == str:
        print x
    else:
        print x.decode("utf-8")
```

UnicodeDecodeError

- <http://wiki.python.org/moin/UnicodeDecodeError>
- http://pyref.infogami.com/__unicode__

Infamous non-breaking Unicode space \xa0

Press CTRL+space / AltGr space on Linux to accidentally create it.

You can't see it. But it breaks everything.

How to fix

Example to how to fix non-breaking space characters which have ended up in reStructuredText `.txt` files. This is Unicode character code A0.

Example `fix_wtf_space.py`:

```
# -*- coding: utf-8 -*-
""" Fix non-breaking space characters which have ended up to reST
.txt files. This is Unicode character code A0.

Press CTRL+space / AltGr space on Linux to accidentally create it.

E.g. as a symptom the following exception is raised if you try
to upload Python egg::

    File "/Library/Python/2.6/site-packages/docutils-0.6-py2.6.egg/docutils/parsers/
→rst/states.py", line 2621, in blank
        self.parent += self.literal_block()
    File "/Library/Python/2.6/site-packages/docutils-0.6-py2.6.egg/docutils/parsers/
→rst/states.py", line 2712, in literal_block
        literal_block = nodes.literal_block(data, data)
    File "/Library/Python/2.6/site-packages/docutils-0.6-py2.6.egg/docutils/nodes.py
→", line 810, in __init__
        TextElement.__init__(self, rawsource, text, *children, **attributes)
    File "/Library/Python/2.6/site-packages/docutils-0.6-py2.6.egg/docutils/nodes.py
→", line 798, in __init__
        textnode = Text(text)
    File "/Library/Python/2.6/site-packages/docutils-0.6-py2.6.egg/docutils/nodes.py
→", line 331, in __new__
        return reprunicode.__new__(cls, data)
```

```
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc2 in position 715: ordinal
↳not in range(128)
"""

import os

def fix(name):
    """ Fix a single .txt file
    """
    input = open(name, "rt")
    text = input.read()
    input.close()
    text = text.decode("utf-8")

    # Show if we get bad hits
    for c in text:
        if c == u"\xa0":
            print "Ufff"

    text = text.replace(u"\xa0", u" ")
    text = text.encode("utf-8")

    output = open(name, "wt")
    output.write(text)
    output.close()

# Process all .txt files in the
# current folder
for f in os.listdir(os.getcwd()):
    if f.endswith(".txt"):
        fix(f)
```

Image troubleshooting

Description

Problems with imaging libraries, image loading and image scaling.

How to test see if your Python Imaging set-up works

Example how to check if Python, Python Imaging Library (PIL) and libjpeg are correctly working together.

Get a sample image:

```
wget http://upload.wikimedia.org/wikipedia/commons/b/bb/JohnCarrollGilbertStuart.jpg
```

Start Python with Zope libraries in PYTHONPATH or Plone debug shell (latter):

```
bin/zopepy

# bin/instance debug # <--- needs Plone site stopped first
```

Run the following on the interactive Python prompt started above:

```
import PIL
from PIL import Image
im = Image.open("JohnCarrollGilbertStuart.jpg") # Open downloaded image
im.thumbnail((64, 64), Image.ANTIALIAS) # See that PIL resize works
im.save("test.jpg") # See that PIL JPEG writing works
```

No Python exceptions should be risen.

Images are not loading

Plone is not loading images or resized images are not available is usually caused by broken PIL installation: PIL used by the Python version that Plone is using does not have proper native libraries (libjpeg etcetera) available to perform imaging operations.

Solution: install the required native libraries for your operating system.

IOError when scaling images on Plone 4

Example:

```
Traceback (most recent call last):
  File "/srv/plone/xxx/plone-new/eggs/plone.app.imaging-1.0.4-py2.6.egg/plone/app/
→imaging/traverse.py", line 73, in createScale
    imgdata, format = field.scale(data, width, height)
  File "/srv/plone/xxx/plone-new/eggs/Products.Archetypes-1.6.6-py2.6.egg/Products/
→Archetypes/Field.py", line 2501, in scale
    image.save(thumbnail_file, format, quality=self.pil_quality)
  File "/srv/plone/python/python-2.6/lib/python2.6/site-packages/PIL-1.1.6-py2.6-
→linux-x86_64.egg/PIL/Image.py", line 1372, in save
    self.load()
  File "/srv/plone/python/python-2.6/lib/python2.6/site-packages/PIL-1.1.6-py2.6-
→linux-x86_64.egg/PIL/ImageFile.py", line 207, in load
    raise IOError(error + " when reading image file")
IOError: decoding error when reading image file
```

This means that libjpeg setup is not working. See above to how to test your set-up.

Installing libraries on Ubuntu / Debian

This applies if you are using system Python to run Plone. Version may vary so `apt-cache search` and `grep` commands are your friends:

```
sudo apt-get install libpng12-dev libjpeg62-dev python-imaging
```

Forcing libjpeg path

Try in `buildout.cfg`:

```
[instance]
...
environment-vars =
    LD_LIBRARY_PATH /srv/plone/python/python-2.6/lib
```

libjpeg.so.8: cannot open shared object file: No such file or directory

On Ubuntu you'll get this error when you try:

```
bin/zopecty
import _imaging
```

Some tips

- <http://stackoverflow.com/questions/5545580/pil-libjpeg-so-8-cannot-open-shared-object-file-no-such-file-or-directory>

Database and transactions troubleshooting

Description

How to debug and fix ZODB database problems in Plone

Introduction

This document contains information to fix and debug ZODB databases with Plone.

BLOBs and POSKeyErrors

The **Plone CMS** from version 4.x onwards stores files and images uploaded to the **ZODB** as blob. They exist in a `var/blobstorage` folder structure on the file system, files being named after (opaque) persistent object ids. When using the default backend, the objects themselves, without file payload, are stored in an append-only database file called *filestorage* and usually the name of this file is `Data.fs`.

If you copy the Plone site database object data (`Data.fs`) and forget to copy the `blobstorage` folder(s), or if data gets out of the sync during the copy, various problems appear on the Plone site:

- You cannot access a content item for which the a corresponding blob file is missing from the file system;
- you cannot rebuild the `portal_catalog` indexes;
- database packing may fail.

Instead, you'll see something like this - an evil `POSKeyError` exception (POS referring to Persistent Object Storage):

```
Traceback (most recent call last):
  File "/fast/xxx/eggs/ZODB3-3.10.3-py2.6-macosx-10.6-i386.egg/ZODB/Connection.py",
↪line 860, in setstate
    self._setstate(obj)
  File "/fast/xxx/eggs/ZODB3-3.10.3-py2.6-macosx-10.6-i386.egg/ZODB/Connection.py",
↪line 922, in _setstate
    obj._p_blob_committed = self._storage.loadBlob(obj._p_oid, serial)
  File "/fast/xxx/eggs/ZODB3-3.10.3-py2.6-macosx-10.6-i386.egg/ZODB/blob.py", line
↪644, in loadBlob
    raise POSKeyError("No blob file", oid, serial)
POSKeyError: 'No blob file'
```

The proper solution to this problem is to:

- Re-copy `blobstorage` folder;

- restart Plone twice in foreground mode (sometimes a freshly copied blobstorage folder does not get picked up - some kind of timestamp issue?). Restarting ZEO clients once seems to be enough.
- *Copy a Plone site*

However you may have failed. You may have damaged or lost your blobstorage forever. To get the Plone site to a working state, all content with bad BLOB data must be deleted (which usually entails losing some site images and uploaded files).

Below is Python code for a *BrowserView* which you can drop in to your own Plone. It creates an admin view which you can call directly via an URL. This code will walk through all the content on your Plone site and try to delete bad content items with BLOBs missing.

The code handles both Archetypes and Dexterity subsystems' content types.

Note: Fixing Dexterity blobs with this code has never been tested - please feel free to update the code in collective.developermanual on GitHub if you find it not working properly.

The code, fixblobs.py:

```
"""
    A Zope command line script to delete content with missing BLOB in Plone, causing
    POSKeyErrors when content is being accessed or during portal_catalog rebuild.

    Tested on Plone 4.1 + Dexterity 1.1.

    http://stackoverflow.com/questions/8655675/cleaning-up-poskeyerror-no-blob-file-
    ↪content-from-plone-site

    Also see:

    https://pypi.python.org/pypi/experimental.gracefulblobmissing/
"""

# Zope imports
from ZODB.POSException import POSKeyError
from zope.component import queryUtility
from Products.CMFCore.interfaces import IPropertiesTool
from Products.CMFCore.interfaces import IFolderish

# Plone imports
from Products.Five import BrowserView
from Products.Archetypes.Field import FileField
from Products.Archetypes.interfaces import IBaseContent
from plone.namedfile.interfaces import INamedFile
from plone.dexterity.content import DexterityContent

def check_at_blobs(context):
    """ Archetypes content checker.

    Return True if purge needed
    """

    if IBaseContent.providedBy(context):
```

```
    schema = context.Schema()
    for field in schema.fields():
        id = field.getName()
        if isinstance(field, FileField):
            try:
                field.get_size(context)
            except POSKeyError:
                print "Found damaged AT FileField %s on %s" % (id, context.
↪absolute_url())
                return True

    return False

def check_dexterity_blobs(context):
    """ Check Dexterity content for damaged blob fields

    XXX: NOT TESTED - THEORETICAL, GUIDELINING, IMPLEMENTATION

    Return True if purge needed
    """

    # Assume dexterity content inherits from Item
    if isinstance(context, DexterityContent):

        # Iterate through all Python object attributes
        # XXX: Might be smarter to use zope.schema introspection here?
        for key, value in context.__dict__.items():
            # Ignore non-contentish attributes to speed up us a bit
            if not key.startswith("_"):
                if INamedFile.providedBy(value):
                    try:
                        value.getSize()
                    except POSKeyError:
                        print "Found damaged Dexterity plone.app.NamedFile %s on %s"
↪% (key, context.absolute_url())
                        return True

    return False

def fix_blobs(context):
    """
    Iterate through the object variables and see if they are blob fields
    and if the field loading fails then poof
    """

    if check_at_blobs(context) or check_dexterity_blobs(context):
        print "Bad blobs found on %s" % context.absolute_url() + " -> deleting"
        parent = context.aq_parent
        parent.manage_delObjects([context.getId()])

def recurse(tree):
    """ Walk through all the content on a Plone site """
    for id, child in tree.contentItems():

        fix_blobs(child)
```

```

        if IFolderish.providedBy(child):
            recurse(child)

class FixBlobs(BrowserView):
    """
    A management view to clean up content with damaged BLOB files

    You can call this view by

    1) Starting Plone in debug mode (console output available)

    2) Visit site.com/@@fix-blobs URL

    """
    def disable_integrity_check(self):
        """ Content HTML may have references to this broken image - we cannot fix_
        ↪ that HTML
        but link integrity check will yell if we try to delete the bad image.

        """
        ptool = queryUtility(IPropertiesTool)
        props = getattr(ptool, 'site_properties', None)
        self.old_check = props.getProperty('enable_link_integrity_checks', False)
        props.enable_link_integrity_checks = False

    def enable_integrity_check(self):
        """ """
        ptool = queryUtility(IPropertiesTool)
        props = getattr(ptool, 'site_properties', None)
        props.enable_link_integrity_checks = self.old_check

    def render(self):
        #plone = getMultiAdapter((self.context, self.request), name="plone_portal_
        ↪ state")
        print "Checking blobs"
        portal = self.context
        self.disable_integrity_check()
        recurse(portal)
        self.enable_integrity_check()
        print "All done"
        return "OK - check console for status messages"

```

Registering the view in ZCML:

```

<browser:view
    for="Products.CMFPlone.interfaces.IPloneSiteRoot"
    name="fix-blobs"
    class=".fixblobs.FixBlobs"
    permission="cmf.ManagePortal"
/>

```

More info

- <http://stackoverflow.com/questions/8655675/cleaning-up-poskeyerror-no-blob-file-content-from-plone-site>
- <https://pypi.python.org/pypi/experimental.gracefulblobmissing/>

Transactions

Transactions are usually problematic only when many ZEO front-end clients are used.

ConflictError

When the site gets more load, `ConflictErrors` start to occur. Zope tries to solve the situation by replaying HTTP requests for `ConflictErrors` and has a default threshold (3) of how many times the request is replayed.

More info

- <https://www.andreas-jung.com/contents/on-zodb-conflict-resolution>

How to debug which object causes ConflictErrors

`ConflictErrors` are caused by concurrent transactions trying to write to the same object(s) - usually `portal_catalog`. They are harmless, but slow down badly coded sites. Plone will retry the HTTP request and transaction three times before giving up.

The OID is visible in the `ConflictError` traceback.

You can turn OID back to the corresponding Python object, as mentioned by A. Jung:

```
from ZODB.utils import p64
app._p_jar[p64(oid)]
```

If every transaction appears as write transaction

If you are not careful, you may accidentally write code which turns all transactions to write transactions. This typically happens when you call some method without realizing that that method eventually modifies a persistent object, causing a database write.

Symptoms:

- Your Undo tab in the Management Interface will be full of entries, one added per page request.
- If you run the server in single Zope server mode, it is slow.
- If you run the server in ZEO mode you get the exceptions like one below. It may happen even with one user. This is because each page load requires more than one HTTP request: HTML load, image load, CSS load and so on. Browser makes many requests per page and those transactions are conflicting, because they are all write transactions.

Traceback example:

```
* Module ZPublisher.Publish, line 202, in publish_module_standard
* Module ZPublisher.Publish, line 170, in publish
* Module ZPublisher.Publish, line 170, in publish
* Module ZPublisher.Publish, line 170, in publish
* Module ZPublisher.Publish, line 157, in publish
* Module plone.app.linkintegrity.monkey, line 15, in zpublisher_exception_hook_
↪ wrapper
* Module ZPublisher.Publish, line 125, in publish
* Module Zope2.App.startup, line 238, in commit
* Module transaction._manager, line 96, in commit
* Module transaction._transaction, line 395, in commit
```



```
* Module transaction._transaction, line 495, in _commitResources
* Module ZODB.Connection, line 510, in commit
* Module ZODB.Connection, line 547, in _commit

ConflictError: database conflict error (oid 0x2b92, class Products.CMFPlone.
↳PropertiesTool.SimpleItemWithProperties)
```

How to debug it

Zope 2 doesn't have many well-documented ZODB debugging tools. Below is one snippet to examine the contents of the last transactions of an offline `Data.fs` file. It is an evolved version of [this original script](#).

- Do something on a badly behaving site.
- Stop Zope instance.
- Run the script below (`debug.py`) on the `Data.fs` file to see what objects have been changed.
- Guess the badly behaving code from the object class name.

Example how to run the script for the last 30 transaction under a Zope egg environment using the `zoepgy` script:

```
bin/zoepgy debug.py -n 30 Data.fs
```

Warning: The following is obsolete with current Zope. `FileIterator` does not take a `pos` argument any more.

Code for `debug.py`:

```
#####
#
# Copyright (c) 2001, 2002 Zope Corporation and Contributors.
# All Rights Reserved.
#
# This software is subject to the provisions of the Zope Public License,
# Version 2.1 (ZPL). A copy of the ZPL should accompany this distribution.
# THIS SOFTWARE IS PROVIDED "AS IS" AND ANY AND ALL EXPRESS OR IMPLIED
# WARRANTIES ARE DISCLAIMED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
# WARRANTIES OF TITLE, MERCHANTABILITY, AGAINST INFRINGEMENT, AND FITNESS
# FOR A PARTICULAR PURPOSE
#
#####
"""Tool to dump the last few transactions from a FileStorage."""

from ZODB.fstools import prev_txn
from ZODB.serialize import ObjectReader, get_refs
from persistent.TimeStamp import TimeStamp
from ZODB.FileStorage.FileStorage import FileIterator
import cStringIO, cPickle
import optparse, getopt
import sys

class Nonce(object): pass

class Reader(ObjectReader):

    def __init__(self):
```

```
        self.identity = None

    def _get_unpickler(self, pickle):
        file = cStringIO.StringIO(pickle)
        unpickler = cPickle.Unpickler(file)
        unpickler.persistent_load = self._persistent_load

        def find_global(modulename, name):
            self.identity = "%s.%s"%(modulename, name)
            return Nonce

        unpickler.find_global = find_global

        return unpickler

    def getIdentity(self, pickle ):
        self.identity = None
        unpickler = self._get_unpickler( pickle )
        unpickler.load()
        return self.identity

    def getObject(self, pickle):
        unpickler = self._get_unpickler( pickle )
        ob = unpickler.load()
        return ob

def pretty_size( size ):
    if size < 1024:
        return "%sB"%(size)
    kb = size / 1024.0
    if kb < 1024.0:
        return '%0.1fKb'%kb
    else:
        mb = kb/1024.0
        return '%0.1fMb'%mb

def run(path, ntxn):
    f = open(path, "rb")
    f.seek(0, 2)

    th = prev_txn(f)
    for i in range(ntxn):
        th = th.prev_txn()
    f.close()
    reader = Reader()
    iterator = FileIterator(path, pos=th._pos)
    for i in iterator:
        print "Transaction ", Timestamp(i.tid), i.user, i.description
        object_types = {}
        for o in i:
            ot = reader.getIdentity(o.data)
            if ot in object_types:
                size, count = object_types[ot]
                object_types[ot] = (size+len(o.data), count+1)
            else:
                object_types[ot] = (len(o.data),1)
```

```

ob = cPickle.loads(o.data)

print "Object data for :" + str(o)

# Not sure why some objects are stored as tuple (object, ())
if type(ob) == tuple and len(ob) == 2:
    ob = ob[0]

if hasattr(ob, "__dict__"):
    for i in ob.__dict__.items():
        if not callable(i[1]):
            print i
else:
    print "can't extract:" + str(ob)

print "-----"

keys = object_types.keys()
keys.sort()
for k in keys:
    # count, class, size (aggregate)
    print " - ", object_types[k][1], k, pretty_size(object_types[k][0])

def main():
    ntxn = 20
    opts, args = getopt.getopt(sys.argv[1:], "n:")
    path, = args
    for k, v in opts:
        if k == '-n':
            ntxn = int(v)
    run(path, ntxn)

if __name__ == "__main__":
    main()

```

zeostorage Client has seen newer transactions than server

If you get:

```
ClientStorageError: zeostorage Client has seen newer transactions than server!
```

, you can fix it by removing `cache-data.zec` from `parts/instance/var/`.

Updating objects created by older code

In the course of development, classes may be renamed or moved. When an object is read from the ZODB, the class required to unpickle the serialized object is named in the pickle data. If this name cannot be imported, you have a broken object on your hands.

In the Zope event log that will show up as, for example:

```

2014-06-19 11:04:04 WARNING OFS.Uninstalled Could not import class
↳ 'ATSimpleStringCriterion' from module 'Products.ATContentTypes.types.criteri
↳ 'ATSimpleStringCriterion'

```

To make the object usable again, the reference needs to be updated to refer to a class that can instantiate this object. One tool that can help you with this is [zodbupdate](#)

In this case, the `ATSimpleStringCriterion` class in question has moved from `Products.ATContentTypes.types.criteria.ATSimpleStringCriterion` to `Products.ATContentTypes.criteria.simplestring`.

To make `zodbupdate` handle this, add a `zodbupdate` entry point to `ATContentTypes`. Depending on your configuration, that may look like this:

```
$ cat ../buildout-cache/eggs/Products.ATContentTypes-2.1.13-py2.7.egg/EGG-INFO/entry_
↪points.txt
[zodbupdate]
renames=Products.ATContentTypes:rename_dict
```

Next, define `rename_dict` in the `__init__.py` of the named package, e.g.:

```
../buildout-cache/eggs/Products.ATContentTypes-2.1.13-py2.7.egg/Products/
↪ATContentTypes/__init__.py
```

In this case, our `rename_dict` will look like this:

```
rename_dict = {
    'Products.ATContentTypes.types.criteria.ATSimpleStringCriterion_
↪ATSimpleStringCriterion':
    'Products.ATContentTypes.criteria.simplestring ATSimpleStringCriterion'}
```

Note: As always, work on a copy of your data first, before working on the live site.

Manually Removing Local Persistent Utilities

Description

This document explains how you can manually remove local persistent utilities that were not properly removed from a product while uninstalling.

Note: Update

There is now a useful tool available, [wildcard.fixpersistentutilities](#), to address these issues TTW (Through The Web). I would suggest trying it before you go through this article.

Purpose

Occasionally you'll download and install a product in Plone that uses local persistent utilities. This usually seems pretty innocent in itself; however, it sometimes happens that when you uninstall the product and remove its egg from the file system, the utility is still registered. This will essentially break your instance unless you make the egg available again so the ZODB can reference the utilities during lookups. This how-to will explain how to remove these utilities manually.

Symptoms

You'll find zope throwing errors like this,

```
AttributeError: type object 'IQueue' has no attribute '__iro__'
```

or

```
AttributeError: type object 'ISalt' has no attribute '__iro__'
```

Prerequisites

You will need appropriate access to the zope server in order to run the site in debug mode.

Step by step

First off, fire up the instance in debug mode

```
./bin/instance debug
```

Get the site manager for your Plone instance. 'app' references the zope root.

```
sm = app.Plone.getSiteManager()
```

Then you'll want to import the guilty utility's interface, unregister it and delete it. It should look somethings like this,

```
from collective.product.interfaces import IUtility, INamedUtility

# for unnamed utility
util = sm.getUtility(IUtility)
sm.unregisterUtility(IUtility)
del util
sm.utilities.unsubscribe(), IUtility)
del sm.utilities.__dict__['_provided'][IUtility]
del sm.utilities._subscribers[0][IUtility]

#also for named utility
util = sm.queryUtility(INamedUtility, name='utility-name')
sm.unregisterUtility(util, INamedUtility, name='utility-name')
del util
del sm.utilities._subscribers[0][INamedUtility]
```

Now you need to commit your changes to the ZODB.

```
import transaction
transaction.commit()
app._p_jar.sync()
```

An Example

I found myself in this situation with the Singing and Dancing product so I'll just go through the code here to fix both a normal utility and named utility found in it.

```
from collective.singing.interfaces import ISalt
from collective.singing.async import IQueue
import transaction

portal = app.Plone
sm = portal.getSiteManager()

util_obj = sm.getUtility(ISalt)
sm.unregisterUtility(provided=ISalt)
del util_obj
sm.utilities.unsubscribe(), ISalt)
del sm.utilities.__dict__['_provided'][ISalt]
del sm.utilities._subscribers[0][ISalt]

util = sm.queryUtility(IQueue, name='collective.dancing.jobs')
sm.unregisterUtility(util, IQueue, name='collective.dancing.jobs')
del util
del sm.utilities._subscribers[0][IQueue]
Handling subscribers, adapters and utilities
sm = app.myportal.getSiteManager()
adapters = sm.utilities._adapters
for x in adapters[0].keys():
    if x.__module__.find("collective.myproduct") != -1:
        print "deleting %s" % x
        del adapters[0][x]
sm.utilities._adapters = adapters

subscribers = sm.utilities._subscribers
for x in subscribers[0].keys():
    if x.__module__.find("collective.myproduct") != -1:
        print "deleting %s" % x
        del subscribers[0][x]
sm.utilities._subscribers = subscribers

provided = sm.utilities._provided
for x in provided.keys():
    if x.__module__.find("collective.myproduct") != -1:
        print "deleting %s" % x
        del provided[x]
sm.utilities._provided = provided

from transaction import commit
commit()
app._p_jar.sync()
```

Removing portal tools

If you still have problems (re)installing products after you removed the broken local persistent components, you probably have to clean the Portal setup tool. You probably see something like this in the error log :

```
setup_tool = app.myportal.portal_setup
toolset = setup_tool.getToolsetRegistry()
if 'portal_myproduct' in toolset._required.keys():
    del toolset._required['portal_myproduct']
    setup_tool._toolset_registry = toolset
```

```
from transaction import commit
commit()
app._p_jar.sync()
```

References

I didn't by any means figure this all out on my own so please do not give me credit for it. Actually, most of this is shamelessly stolen. Thanks for the original fixers of the problem! Here are my references:

- <http://blog.fourdigits.nl/removing-a-persistent-local-utility>
- <http://blog.fourdigits.nl/removing-a-persistent-local-utility-part-ii>

Automating Plone Deployment

Developing for Plone

An overview of all documentation for developers, both for writing your own add-ons and for working with Plone itself.

Develop Plone Add ons

To develop an add-on, you need a package to put your code in, plus ways to make it interact with Plone itself and the user. And a way to release your package to your audience.

In short:

Create a package

With the help of Mr.Bob and templates for plone, that is quickly done:

bobtemplates introduction

`bobtemplates.plone` provides a `mr.bob` template to generate packages for Plone projects.

To create a package like `collective.myaddon`:

```
$ mrbob -O collective.myaddon bobtemplates:plone_addon
```

You can also create a package with nested namespace:

```
$ mrbob -O collective.foo.myaddon bobtemplates:plone_addon
```

Options

On creating a package you can choose from the following options. The default value is in [square brackets]:

Package Type? [Basic] Options are Basic, Dexterity and Theme.

Author's name Should be something like 'John Smith'.

Author's email Should be something like 'john@plone.org'.

Author's github username Should be something like 'john'.

Package description [An add-on for Plone] One-liner describing what this package does. Should be something like 'Plone add-on that ...'.

Plone version [4.3.4] Which Plone version would you like to use?

Add example view? [True] Do you want to register a browser view 'demoview' as an example?

Features

Package created with `bobtemplates.plone` use the current best-practices when creating an addon.

Buildout The package is contained in a buildout that allows you to build Plone with the new package installed for testing-purposes.

Tests The package comes with a test setup and some `tests` for installing the package. It also contains a `robot-test` that tests logging in. The buildout also contains a config to allow testing the package on `travis` that sends `notifications` by email to the package autor.

Profile The package contains a `Generic Setup Profile` that installs a browserlayer.

Locales The package registers a directory for locales.

Template-Overrides The package registers the folder `browser/overrides` as a directory where you can drop template-overrides using `z3c.jbot`.

Setuphandler The package contains a `setuptools.py` where you can add code that is executed on installing the package.

Compatibility

Addons created with `bobtemplates.plone` are tested to work in Plone 4.3.x and Plone 5. They should also work with older versions but that was not tested.

Installation

Use in a buildout

```
[buildout]
parts += mrbob

[mrbob]
recipe = zc.recipe.egg
eggs =
    mr.bob
    bobtemplates.plone
```

This creates a `mrbob-executable` in your `bin`-directory. Call it from the `src`-directory of your Plone project like this.:

```
$ ../bin/mrbob -O collective.foo bobtemplates:plone_addon
```

Installation in a virtualenv

You can also install `bobtemplates.plone` in a virtualenv.:

```
$ pip install mr.bob
$ pip install bobtemplates.plone
```

Now you can use it like this:

```
$ mrbob -O collective.foo bobtemplates:plone_addon
```

See [mr.bob](#) documentation for further information.

Develop with Dexterity

Dexterity is covered in detail in the Dexterity Developer Manual, which includes an extensive tutorial on setting up a Dexterity development environment.

Upgrading to Plone 5.1

Upgrade a custom add-on to Plone 5.1

Installation code

See [PLIP 1340](#) for a discussion of this change.

From CMFQuickInstallerTool to GenericSetup

The add-ons control panel in Plone 5.1 no longer supports installation or uninstallation code in `Extensions/install.py` or `Extensions/Install.py`. If you have such code, you must switch to a `GenericSetup` profile.

`GenericSetup` is already the preferred way of writing installation code since Plone 3. If you must use the old way, you can still use the `portal_quickinstaller` in the Management Interface.

In a lot of cases, you can configure `xml` files instead of using Python code. In other cases you may need to write custom installer code (`setuptools.py`). See the [GenericSetup documentation](#).

default profile

Historically, when your add-on had multiple profiles, their names would be sorted alphabetically and the first one would be taken as the installation profile. It was always recommended to use `default` as name of this first profile.

Since Plone 5.1, when there is a `default` profile, it is always used as the installation profile, regardless of other profile names. Exception: when this `default` profile is marked in an `INonInstallable` utility, it is ignored and Plone falls back to using the first from the alphabetical sorting.

Uninstall

An uninstall profile is not required, but it is highly recommended.

Until Plone 5.0 the CMFQuickInstallerTool used to do an automatic partial cleanup, for example removing added skins and css resources. This was always only partial, so you could not rely on it to fully cleanup the site.

Since Plone 5.1 this cleanup is no longer done. Best practice is to create an uninstall profile for all your packages.

If you were relying on this automatic cleanup, you need to add extra files to clean it up yourself. You need to do that when your default profile contains one of these files:

- `actions.xml`
- `componentregistry.xml`
- `contenttyperegistry.xml`. This seems rarely used. Note: the `contenttyperegistry` import step only supports adding, not removing. You may need to improve that code based on the old `CMFQuickInstallerTool` code.
- `cssregistry.xml`
- `jsregistry.xml`
- `skins.xml`
- `toolset.xml`
- `types.xml`
- `workflows.xml`

When there is no uninstall profile, the add-ons control panel will give a warning. An uninstall profile is a profile that is registered with the name `uninstall`. For an example, see <https://github.com/plone/plone.app.multilingual/tree/master/src/plone/app/multilingual/profiles/uninstall>

Don't use `portal_quickinstaller`

Old code:

```
qi = getToolByName(self.context, name='portal_quickinstaller')
```

or:

```
qi = self.context.portal_quickinstaller
```

or:

```
qi = getattr(self.context, 'portal_quickinstaller')
```

or:

```
qi = getUtility(IQuickInstallerTool)
```

New code:

```
from Products.CMFPlone.utils import get_installer
qi = get_installer(self.context, self.request)
```

or if you do not have a request:

```
qi = get_installer(self.context)
```

Alternatively, since it is a browser view, you can get it like this:

```
qi = getMultiAdapter((self.context, self.request), name='installer')
```

or with `plone.api`:

```
from plone import api
api.content.get_view(
    name='installer',
    context=self.context,
    request=self.request)
```

If you need it in a page template:

```
tal:define="qi context/@@installer"
```

Warning: Since the code really does different things than before, the method names were changed and they may accept less arguments or differently named arguments.

Products namespace

There used to be special handling for the Products namespace. Not anymore.

Old code:

```
qi.installProduct('CMFPlacefulWorkflow')
```

New code:

```
qi.install_product('Products.CMFPlacefulWorkflow')
```

isProductInstalled

Old code:

```
qi.isProductInstalled(product_name)
```

New code:

```
qi.is_product_installed(product_name)
```

installProduct

Old code:

```
qi.installProduct(product_name)
```

New code:

```
qi.install_product (product_name)
```

Note that no keyword arguments are accepted.

installProducts

This was removed. You should iterate over a list of products instead.

Old code:

```
product_list = ['package.one', 'package.two']
qi.installProducts (product_list)
```

New code:

```
product_list = ['package.one', 'package.two']
for product_name in product_list:
    qi.install_product (product_name)
```

uninstallProducts

Old code:

```
qi.uninstallProducts ([product_name])
```

New code:

```
qi.uninstall_product (product_name)
```

Note that we only support passing one product name. If you want to uninstall multiple products, you must call this method multiple times.

reinstallProducts

This was removed. Reinstalling is usually not a good idea: you should use an upgrade step instead. If you need to, you can uninstall and install if you want.

getLatestUpgradeStep

Old code:

```
qi.getLatestUpgradeStep (profile_id)
```

New code:

```
qi.get_latest_upgrade_step (profile_id)
```

upgradeProduct

Old code:

```
qi.upgradeProduct (product_id)
```

New code:

```
qi.upgrade_product (product_id)
```

isDevelopmentMode

This was a helper method that had got nothing to with the quick installer.

Old code:

```
qi = getToolByName(aq_inner(self.context), 'portal_quickinstaller')
return qi.isDevelopmentMode()
```

New code:

```
from Globals import DevelopmentMode
return bool(DevelopmentMode)
```

Note: The new code works already since Plone 4.3.

All deprecated methods

Some of these were mentioned already.

Some methods are no longer supported. These methods are still there, but they do nothing:

- listInstallableProducts
- listInstalledProducts
- getProductFile
- getProductReadme
- notifyInstalled
- reinstallProducts

Some methods have been renamed. The old method names are kept for backwards compatibility. They do roughly the same as before, but there are differences. And all keyword arguments are ignored. You should switch to the new methods instead:

- isProductInstalled, use is_product_installed instead
- isProductInstallable, use is_product_installable instead
- isProductAvailable, use is_product_installable instead
- getProductVersion, use get_product_version instead
- upgradeProduct, use upgrade_product instead

- `installProducts`, use `install_product` with a single product instead
- `installProduct`, use `install_product` instead
- `uninstallProducts`, use `uninstall_product` with a single product instead.

INonInstallable

There used to be one `INonInstallable` interface in `CMFPlone` (for hiding profiles) and another one in `CMFQuickInstallerTool` (for hiding products). In the new situation, these are combined in the one from `CMFPlone`.

Sample usage:

In `configure.zcml`:

```
<utility factory=".setuphandlers.NonInstallable"
  name="your.package" />
```

In `setuphandlers.py`:

```
from Products.CMFPlone.interfaces import INonInstallable
from zope.interface import implementer

@implementer(INonInstallable)
class NonInstallable(object):

    def getNonInstallableProducts(self):
        # (This used to be in CMFQuickInstallerTool.)
        # Make sure this package does not show up in the add-ons
        # control panel:
        return ['collective.hidden.package']

    def getNonInstallableProfiles(self):
        # (This was already in CMFPlone.)
        # Hide the base profile from your.package from the list
        # shown at site creation.
        return ['your.package:base']
```

When you do not need them both, you can let the other return an empty list, or you can leave that method out completely.

Note: If you need to support older Plone versions at the same time, you can let your class implement the old interface as well:

```
from Products.CMFQuickInstallerTool.interfaces import (
    INonInstallable as INonInstallableProducts)

@implementer(INonInstallableProducts)
@implementer(INonInstallable)
class NonInstallable(object):
    ...
```


Retina image scales

In the Image Handling Settings control panel in Site Setup, you can configure Retina mode for extra sharp images. When you enable this, it will result in image tags like this, for improved viewing on Retina screens:

```

```

To benefit from this new feature, you must use the `tag` method of image scales:

```
<img tal:define="images obj/@@images"
      tal:replace="structure python:images.scale('image', scale='tile').tag(css_class=
      ↪ 'image-tile') ">
```

If you are iterating over a list of image brains, you should use the new `@@image_scale` view of the portal or the navigation root. This will cache the result in memory, which avoids waking up the objects the next time.

```
<tal:block define="image_scale portal/@@image_scale">
  <tal:results tal:repeat="brain batch">
    <img tal:replace="structure python:image_scale.tag(item, 'image', scale='tile
    ↪ ', css_class='image-tile') ">
  </tal:results>
</tal:block>
```

Add your package to buildout

Edit your `buildout.cfg` file to add the package to your `egg` list and your `develop` list. Run `buildout`.

The Plone Collective

This is an organization for developers of Plone add-ons to work collectively. Software that is released in here follows a simple, collaborative model: every member can contribute to every project.

This means you will have the best chance of having other people contributing to your add-on. When your add-on is generic enough to be useful to other people, please consider to release it here.

[Read more on how to become a member of the Plone Collective](#)

Releasing your package

Releasing an addon

Your addon should be listed and hosted on PyPI if you want other people to use your addon.

Warning: Everything on PyPI is public. Be careful not to hard-code passwords in *any* file.

Setup necessary packages

To setup all needed packages you need to run the following command.

```
pip install zest.releaser zest.pocompile check-manifest
```

This takes care of everything you should do: - Check if all files will be in the package. - Set the version number - Tag the release - Compile any .mo file to .po files - Make the actual release - Bump the version.

Note: This installs the packages into your global python installation. An alternative would be installing the packages in a *virtualenv*.

Releasing a package

Use the `fullrelease` command in the root of your checkout.

```
$ fullrelease
```

See also:

how to use virtualenv controlled non-system wide Python

Full zest.releaser documentation <http://zestreleaser.readthedocs.org/en/latest/>

plone.api coding conventions“

<http://opensourcehacker.com/2012/08/14/high-quality-automated-package-releases-for-python-with-zest-releaser/>

Working with JavaScript

Note: Working with JavaScript has changed considerably in Plone 5. Read the note at the beginning of the document.

JavaScript

Description

Writing, including and customizing JavaScript for Plone add-ons

Note: This part of the documentation is **under construction**.

Important parts for adding JavaScript to add-ons can be found in the documentation about the *Plone 5 Resource Registry*.

While we are updating this documentation, you should also look at the following blog posts:

- [Updating JavaScript for Plone 5](#)
- [Customizing JavaScript pattern options in Plone 5](#)

Introduction

Description

Writing, including and customizing JavaScript for Plone add-ons

Note: This part of the documentation is **under construction**. Important parts for adding JavaScript to addons can be found in the documentation about the *Plone 5 Resource Registry*. You should also look at the JavaScript part of the official [plone training](#). While we are updating this documentation, you should also look at the following blogposts:

- [Updating JavaScript for Plone 5](#)
- [Customizing JavaScript pattern options in Plone 5](#)

For JavaScript and CSS development, Plone 5 makes extensive use of tools like Bower, Grunt, RequireJS and Less for an optimized development process. The JavaScript and CSS resources are managed in the *Plone 5 Resource Registry*. The Resource Registry was completely rewritten in Plone 5 to support the new dependency based RequireJS approach. It also allows us to build Less and RequireJS bundles Through-The-Web for a low entry barrier.

JavaScript basic tips

Always use DOM ready event before executing your DOM manipulation.

Don't include JavaScript inline in HTML code unless you are passing variables from Python to JavaScript.

Use JSLint with your code editor and ECMAScript 5 strict mode to catch common JavaScript mistakes (like missing var).

For more JavaScript tips see [brief introduction to good JavaScript practices and JSLint](#)

Add a pattern or other javascript to your own bundle

1. Write your JavaScript file or pattern

There are two options:

- Use `define` to define a pattern that can be imported by other modules
- Use `require` to add your own javascript that does not need to be imported by other modules (see here for more details: <https://github.com/plone/Products.CMFPlone/issues/1163#issuecomment-148086841>)

A good example for a new pattern can be found here: <https://github.com/collective/pat-fancytree> (Work in progress)

It uses the new Base Pattern that was added to patternslib recently. The way how this example registers the Pattern assumes that `require.js` is available and used. To make a pattern work without `require.js` please use the way like it is proposed by the patternslib documentation: <http://patternslib.com/creating-a-pattern/#main-content>

There is also a Yeoman patternslib generator that can be found here: <https://www.npmjs.com/package/generator-patternslib> It produces a boilerplate for creating a new pattern

2. Add your pattern or JavaScript file as a resource (in registry.xml)

```
<records prefix="plone.resources/MY-PATTERN"
        interface='Products.CMFPlone.interfaces.IResourceRegistry'>
```

```
<value key="js">++resource+MY-PATTERN-PATH/MY-PATTERN.js</value>
</records>
```

Make sure that your pattern is available over the provided resource directory path

3. Create bundle resource (only if “define” is used under 1)

See `Products.CMFPlone/Products/CMFPlone/static/plone.js` for an example

4. Add bundle resource (only if “define” is used under 1) (in `registry.xml`)

```
<records prefix="plone.resources/MY-BUNDLE-RESOURCE"
  interface='Products.CMFPlone.interfaces.IResourceRegistry'>
  <value key="js">++resource++MY-BUNDLE-RESOURCE-PATH/MY-BUNDLE-RESOURCE.js</value>
  <value key="css">
    <element>++resource++MY-BUNDLE-RESOURCE-PATH/MY-BUNDLE-RESOURCE.less</element>
  </value>
</records>
```

5. Define bundle

```
<records prefix="plone.bundles/MY-BUNDLE"
  interface='Products.CMFPlone.interfaces.IBundleRegistry'>
  <value key="resources">
    <element>MY-BUNDLE-RESOURCE</element>
  </value>
  <value key="enabled">True</value>
  <value key="jscompilation">++resource++MY-BUNDLE-PATH/MY-BUNDLE-compiled.min.js</
  <value>
  <value key="csscompilation">++resource++MY-BUNDLE-PATH/MY-BUNDLE-compiled.min.css</
  <value>
  <value key="depends">BUNDLE-DEPENDENCY</value>
</records>
```

6. Compile bundle

First you need to install your addon in a fresh plone site. Then execute

```
bin/plone-compile-resources --site-id=Plone --bundle=MY-BUNDLE
```

Open questions for addons developers:

- Do I need to create a bundle for every add on? I there a possibility to add a resource to an existing bundle? If yes, how is this done?

What is missing here?

- How do I setup a dev environment for the JavaScript topic?

How to add a patternslib pattern to plone bundle in `Products.CMFPlone`

1. Add resource

```
<records prefix="plone.resources/patternslib-patterns-autofocus"
  interface='Products.CMFPlone.interfaces.IResourceRegistry'>
  <value key="js">++plone++static/components/patternslib/src/pat/autofocus/
  <value>
  </value>
</records>
```

2. Add pattern to `static/plone.js`

AJAX

Description

Creating AJAX programming logic in Plone.

Introduction

‘AJAX <http://en.wikipedia.org/wiki/Ajax_%28programming%29>’_ (an acronym for Asynchronous JavaScript and XML) is a group of interrelated web development techniques used on the client-side to create asynchronous web applications.

JSON views and loading data via AJAX

The best way to output JSON for AJAX call endpoints is to use Python’s dict structure and convert it to JSON using Python `json.dumps()` call.

You should pass the AJAX target URLs to your JavaScript using the settings passing pattern explained above.

Examples

Generator

- <https://github.com/miohtama/silvuple/blob/master/silvuple/views.py#L342>

AJAX loader

- <https://github.com/miohtama/silvuple/blob/master/silvuple/static/main.js#L247>

Cross-Origin Resource Sharing (CORS) proxy view

Old web browsers do not support [Allow-acces-origin HTTP header](#) needed to do cross-domain AJAX requests (IE6, IE7).

Below is an example how to work around this for jQuery `getJSON()` calls by

- Detecting browsers which do not support this using jQuery.support API
- Doing an alternative code path through a local website proxy view which uses Python `urllib` to make server-to-server call and return it as it would be a local call, thus working around cross-domain restriction

This example is for Plone, but the code is portable to other web frameworks.

Note: This is not a full example code. Basic Python and JavaScript skills are needed to interpret and adapt the code for your use case.

JavaScript example

```
/**
 * Call a RESTful service via AJAX
 *
 * The final URL is constructed by REST function name, based
 * on a base URL from the global settings.
 *
 * If the browser does not support cross domain AJAX calls
 * we'll use a proxy function on the local server. For
 * performance reasons we do this only when absolutely needed.
 *
 * @param {String} functionName REST function name to a call
 *
 * @param {Object} Arguments as a dictionary like object, passed to remote call
 */
function callRESTful(functionName, args, callback) {

    var src = myoptions.restService + "/" + functionName;

    // set to true to do proxied request on every browser
    // useful if you want to use Firebug to debug your server-side proxy view
    var debug = false;

    console.log("Doing remote call to:" + src)

    // We use jQuery API to detect whether a browser supports cross domain AJAX
    ↪calls
    // http://api.jquery.com/jquery.support/
    if(!jQuery.support.cors || debug) {
        // http://alexn.org/blog/2011/03/24/cross-domain-requests.html
        // Opera 10 doesn't have this feature, neither do IEExplorer < 8,
    ↪Firefox < 3.5

        console.log("Mangling getJSON to go through a local proxy")

        // Change getJSON to go to our proxy view on a local server
        // and pass the original URL as a parameter
        // The proxy view location is given as a global JS variable
        args.url = src;
        src = myoptions.portalUrl + "/@@proxy";
    }

    // Load data from the server
    $.getJSON(src, args, function(data) {
        // Parse incoming data and construct Table rows according to it
        console.log("Data successfully loaded");
        callback(data, args);
    });
}
```

The server-side view:

```
import socket
import urllib
import urllib2
from urllib2 import HTTPError
```

```

from Products.Five import BrowserView
from mysite.app import options

class Proxy(BrowserView):
    """
    Pass a AJAX call to a remote server. This view is mainly indended to be used
    with jQuery.getJSON() requests.

    This will work around problems when a browser does not support Allow-Access-
    ↪Origin HTTP header (IE).

    Asssuming only HTTP GET requests are made.s
    """

    def isAllowed(self, url):
        """
        Check whether we are allowed to call the target URL.

        This prevents using your service as an malicious proxy
        (to call any internet service).
        """

        allowed_prefix = options.REST_SERVICE_URL

        if url.startswith(allowed_prefix):
            return True

        return False

    def render(self):
        """
        Use HTTP GET ``url`` query parameter for the target of the real request.
        """

        # Make sure any theming layer won't think this is HTML
        # http://stackoverflow.com/questions/477816/the-right-json-content-type
        self.request.response.setHeader("Content-type", "application/json")

        url = self.request.get("url", None)
        if not url:
            self.request.response.setStatus(500, "url parameter missing")

        if not self.isAllowed(url):
            # The server understood the request, but is refusing to fulfill it.
            ↪Authorization will not help and the request SHOULD NOT be repeate
            self.request.response.setStatus(403, "proxying to the target URL not
            ↪allowed")
            return

        # Pass other HTTP GET query parameters directly to the target server
        params = {}
        for key, value in self.request.form.items():
            if key != "url":
                params[key] = value

        # http://www.voidspace.org.uk/python/articles/urllib2.shtml
        data = urllib.urlencode(params)

```

```
full_url = url + "?" + data
req = urllib2.Request(full_url)

try:

    # Important or if the remote server is slow
    # all our web server threads get stuck here
    # But this is UGLY as Python does not provide per-thread
    # or per-socket timeouts thru urllib
    original_timeout = socket.getdefaulttimeout()
    try:
        socket.setdefaulttimeout(10)

        response = urllib2.urlopen(req)
    finally:
        # restore original timeout
        socket.setdefaulttimeout(original_timeout)

    # XXX: How to stream response through Zope
    # AFAIK - we cannot do it currently

    return response.read()

except HTTPError, e:
    # Have something more useful to log output as plain urllib exception
    # using Python logging interface
    # http://docs.python.org/library/logging.html
    logger.error("Server did not return HTTP 200 when calling remote proxy_
↪URL:" + url)
    for key, value in params.items():
        logger.error(key + ": " + value)

    # Print the server-side stack trace / error page
    logger.error(e.read())

    raise e
```

Registering the view in ZCML:

```
<browser:view
    for="Products.CMFPlone.interfaces.IPloneSiteRoot"
    name="proxy"
    class=".views.Proxy"
    permission="zope.Public"
/>
```

Speeding up AJAX loaded content HTML

By observing Plone's `main_template.pt`, having a `True` value on the `ajax_load` request key means some parts of the page aren't displayed, hence the speed:

- No CSS or JavaScript from `<head />` tag is loaded
- Nothing from the `plone.portaltop` ViewletManager, such as the personal bar, searchbox, logo and main menu

- Nothing from the `plone.portalfooter` ViewletManager, which contains footer and colophon information, site actions and the Analytics javascript calls if you have that configured in your site
- Neither the left nor the right column, along with all the portlets there assigned

Background

Component architecture

Introduction

Plone logic is wired together by Zope 3 component architecture. It provides “enterprise business logic” engine for Plone.

The architecture provides pluggable system *interfaces*, adapters, utilities and registries. The wiring of components is done on XML based language called *ZCML*.

Database drops using Generic setup

Zope 3 components act on Python codebase level which is shared by all sites in the same Zope application server process. When you install new add-ons to Plone site, the add-ons modify the site database using *GenericSetup* framework. GenericSetup is mostly visible as */profiles/default* folder and its XML files in your add-on.

More info

- <http://www.muthukadan.net/docs/zca.html>

Interfaces

Introduction

Interfaces define what methods an object provides. Plone extensively uses interfaces to define APIs between different subsystems. They provide a more consistent and declarative way to define bridges between two different things, when duct taping is not enough.

An interface defines the shape of a hole where different pieces fit. The shape of the piece is defined by the interface, but the implementation details like color, material, etc. can vary.

See `zope.interface` package [README](#).

Common interfaces

Some interfaces are commonly used throughout Plone.

The usual use case is that a *context directive for a view* is provided, specifying where the view is available (e.g. for which content types).

`zope.interface.Interface` Base class of all interfaces. Also used as a `*` wildcard when registering views, meaning that the view applies on every object.

`Products.CMFCore.interfaces.IContentish` All *content* items on the site. In the site root, this interface excludes Zope objects like `acl_users` (the user folder) and `portal_skins` which might otherwise appear in the item listing when you iterate through the root content.

Products.CMFCore.interfaces.IFolderish All *folders* in the site.

Products.CMFCore.interfaces.ISiteRoot The Plone site root object.

plone.app.layout.navigation.interfaces import INavigationRoot Navigation top object - where the breadcrumbs are anchored. On multilingual sites, this is the top-level folder for the current language.

Implementing one or multiple interfaces

Use `zope.interface.implements()` in your class body. Multiple interfaces can be provided as arguments.

Example:

```
from zope.interface import implements
from collective.mountpoint.interfaces import ILocalSyncedContent
from ora.objects.interfaces import IORAResearcher

class MyContent(folder.ATFolder):
    """A Researcher synchronized from ORA"""
    implements(IORAResearcher, ILocalSyncedContent)
```

Removing parent class interface implementations

`implementsOnly()` redeclares all inherited interface implementations. This is useful if you, for example, want to make *z3c.form* widget bindings more accurate.

Example:

```
zope.interface.implementsOnly(IAddressWidget)
```

Checking whether object provides an interface

`providedBy`

In Python you can use code:

```
from yourpackage.interfaces import IMyInterface

if IMyInterface.providedBy(object):
    # do stuff
else:
    # was not the kind of object we wanted
```

`plone_interface_info`

In page templates you can use `plone_interface_info` helper view:

```
<div tal:define="iinfo context/@@plone_interface_info">
  <span tal:condition="python:iinfo.provides('your.dotted.interface.IName')">
    Do stuff requiring your interface.
  </span>
</div>
```

See also

- <https://github.com/plone/plone.app.layout/blob/master/plone/app/layout/globals/interface.py>

Interface resolution order

Interface resolution order (IRO) is the list of interfaces provided by the object (directly, or implemented by a class), sorted by priority.

Interfaces are evaluated from zero index (highest priority) to the last index (lowest priority).

You can access this information for the object for debugging purposes using a magical attribute:

```
object.__provides__.__iro__.
```

Note: Since adapter factories are *dynamic* (adapter interfaces not hardcoded on the object), the object can still adapt to interfaces which are not listed in `__iro__`.

Getting interface string id

The interface id is stored in the `__identifier__` attribute.

Example file `yourpackage/interfaces.py`:

```
import zope.interface

class IFoo(zope.interface.Interface):
    pass

# id is yourpackage.interfaces.IFoo
id = IFoo.__identifier__
```

Note that this attribute does not respect import aliasing.

Example: `Products.ATContentTypes.interfaces.IATDocument.__identifier__` is `Products.ATContentTypes.interfaces.document.IATDocument`.

Getting interface class by its string id

Use the `zope.dottedname` package.

Example:

```
import zope.interface
from zope.dottedname.resolve import resolve

class IFoo(zope.interface.Interface):
    pass

# id is yourpackage.interfaces.IFoo
id = IFoo.__identifier__
interface_class == resolve(id)
assert IFoo == interface_class
```

Applying interfaces for several content types

You can apply marker interfaces to content types at any time.

Example use cases:

- You want to assign a viewlet to a set of particular content types.
- You want to enable certain behavior on certain content types.

Note: A marker interface is needed only when you need to create a common nominator for several otherwise unrelated classes. You can use one existing class or interface as a context without explicitly creating a marker interface. Places accepting `zope.interface.Interface` as a context usually accept a normal Python class as well (isinstance behavior).

You can assign the marker interface for several classes in ZCML using a `<class>` declaration. Here we're assigning `ILastModifiedSupport` to documents, events and news items:

```
<!-- List of content types where "last modified" viewlet is enabled -->
<class class="Products.ATContentTypes.content.document.ATDocument">
  <implements interface=".interfaces.ILastModifiedSupport" />
</class>

<class class="Products.ATContentTypes.content.event.ATEvent">
  <implements interface=".interfaces.ILastModifiedSupport" />
</class>

<class class="Products.ATContentTypes.content.newsitem.ATNewsItem">
  <implements interface=".interfaces.ILastModifiedSupport" />
</class>
```

Then we can have a view for these content types only using the following:

```
.. code-block:: python
```

```
from Products.Five import BrowserView
from interfaces import ILastModifiedSupport
from plone.app.layout.viewlets.interfaces import IBelowContent
```

```
class LastModified(BrowserView): """ View for .interfaces.ILastModifiedSupport only """
```

```
<browser:view
    for=".interfaces.ILastModifiedSupport"
    name="lastmodified"
    class=".views.LastModified"
    template="templates/lastmodified.pt"
/>
```

Related:

- `zope.dottedname` allows you to resolve dotted names to Python objects manually

Dynamic marker interfaces

Zope allows you to dynamically turn on and off interfaces on any content objects through the Management Interface. Browse to any object and visit the *Interfaces* tab.

Marker interfaces might need to be explicitly declared using the *ZCML* `<interface>` directive, so that Zope can find them:

```
<!-- Declare marker interface, so that it is available in the Management Interface -->
<interface interface="mfabrik.app.interfaces.promotion.IPromotionsPage" />
```

Note: The interface dotted name must refer directly to the interface class and not to an import from other module, like `__init__.py`.

Setting dynamic marker interfaces programmatically

Use the `mark()` function from `Products.Five`.

Example:

```
from Products.Five.utilities.marker import mark

mark(portal.doc, interfaces.IBuyableMarker)
```

Note: This marking persists with the object: it is not temporary.

Under the hood: `mark()` delegates to `zope.interface.directlyProvides()` — with the result that a persistent object (e.g. content item) has a reference to the interface class you mark it with in its `__provides__` attribute; this attribute is serialized and loaded by ZODB like any other reference to a class, and `zope.interface` uses object specification descriptor magic (just like it does for any other object, persistent or not) to resolve provided interfaces.

To remove a marker interface from an object, use the `erase()` function from `Products.Five`.

Example:

```
from Products.Five.utilities.marker import erase

erase(portal.doc, interfaces.IBuyableMarker)
```

Tagged values

Tagged values are arbitrary metadata you can stick on `zope.interface.Interface` subclasses. For example, the `plone.autoform` package uses them to set form widget hints for `zope.schema` data model declarations.

Adapters

Introduction

Adapters make it possible to extend the behavior of a class without modifying the class itself. This allows more modular, readable code in complex systems where there might be hundreds of methods per class. Some more advantages of this concept are:

- The class interface itself is more readable (less visible clutter);

- class functionality can be extended outside the class source code;
- add-on products may extend or override parts of the class functionality. Frameworks use adapters extensively, because adapters provide easy integration points. External code can override adapters to retrofit/modify functionality. For example: a theme product might want to override a searchbox viewlet to have a search box with slightly different functionality and theme-specific goodies.

The downside is that adapters cannot be found by “exploring” classes or source code. They must be well documented in order to be discoverable.

Read more about adapters in the [zope.component README](#).

Adapter ZCML.

Adapters are matched by:

- Provider interface (what functionality adapter provides).
- Parameter interfaces.

There are two kinds of adapters:

- Normal adapters that take only one parameter.
- Multi-adapters take many parameters in the form of a tuple.

Example adapters users

- [Theme specific adapters](#)

Registering an adapter

Registering using ZCML

An adapter provides functionality to a class. This functionality becomes available when the interface is queried from the instance of class.

Below is an example how to make a custom “image provider”. The image provider provides a list of images for arbitrary content.

This is the image provider interface:

```
from zope.interface import Interface

class IProductImageProvider(Interface):

    def getImages(self):
        """ Get Images associated with the product.

        @return: iterable of Image objects
        """
```

This is our content class:

```
class MyShoppableItemType(folder.ATFolder):
    """ Buyable physical good with variants of title and price and multiple images
    """
    implements(IVariantProduct)
```

```
meta_type = "VariantProduct"
schema = VariantProductSchema
```

This is the adapter for the content class:

```
import zope.interface

from getpaid.variantsproduct.interfaces.multiimageproduct import IProductImageProvider

class FolderishProductImageProvider(object):
    """ Mix-in class which provide product image management functions.

    Assume the content itself is a folderish archetype content type and
    all contained image objects are product images.
    """

    zope.interface.implements(IProductImageProvider)

    def __init__(self, context):
        # Each adapter takes the object itself as the construction
        # parameter and possibly provides other parameters for the
        # interface adaption
        self.context = context

    def getImages(self):
        """ Return a sequence of images.

        Perform folder listing and filter image content from it.
        """

        images = self.context.listFolderContents(
            contentFilter={"portal_type" : "Image"})

        return images
```

Register the adapter for your custom content type `MyShoppableItemType` in the `configure.zcml` file of your product:

```
<adapter
  for=".shop.MyShoppableItemType"
  provides=".interfaces.IProductImageProvider"
  factory=".images.FolderishProductImageProvider"
/>
```

Then we can query the adapter and use it. Unit testing example:

```
def test_get_images(self):
    self.loginAsPortalOwner()
    self.portal.invokeFactory("MyShoppableItemType", "product")
    product = self.portal.product
    image_provider = IProductImageProvider(product)
    images = image_provider.getImages()

    # Not yet any uploaded images
    self.assertEqual(len(images), 0)
```

Registering using Python

Register to *Global Site Manager* using `registerAdapter()`.

Example:

```
from zope.component import getGlobalSiteManager

layer = klass.layer

gsm = getGlobalSiteManager()
gsm.registerAdapter(factory=MyClass, required=(layer,),
                    name=klass.__name__, provided=IWidgetDemo)

return klass
```

More info

- <http://www.muthukadan.net/docs/zca.html#registration>

Generic adapter contexts

The following interfaces are useful when registering adapters:

zope.interface.Interface Adapts to any object

Products.CMFCore.interfaces.IContentish Adapts to any Plone content object

zope.publisher.interfaces.IBrowserView Adapts to any `BrowserView(context, request)` object

Multi-adapter registration

You can specify any number of interfaces in the `<adapter for="" />` attribute. Separate them with spaces or newlines.

Below is a view-like example which registers against:

- any context (`zope.interface.Interface`);
- HTTP request objects (`zope.publisher.interfaces.browser.IBrowserRequest`).

Emulate view registration (context, request):

```
<adapter
  for="zope.interface.Interface
      zope.publisher.interfaces.browser.IBrowserRequest"
  provides="gomobile.mobile.interfaces.IMobileTracker"
  factory=".bango.BangoTracker"
/>
```

Getting the adapter

There are two functions that may be used to get an adapter:

- `zope.component.getAdapter` will raise an exception if the adapter is not found.
- `zope.component.queryAdapter` will return `None` if the adapter is not found.

getAdapter/queryAdapter arguments:

Tuple consisting of: (*Object implementing the first interface, object implementing the second interface, ...*) The interfaces are in the order in which they were declared in the <adapter for=""> attribute.

Adapter marker interface.

Example registration:

```
<!-- Register header animation picking logic - override this for your custom logic -->
<adapter
  provides="plone.app.headeranimation.interfaces.IHeaderAnimationPicker"
  for="plone.app.headeranimation.behaviors.IHeaderBehavior
    Products.CMFCore.interfaces.IContentish
    zope.publisher.interfaces.browser.IBrowserRequest"
  factory=".picker.RandomHeaderAnimationPicker"
/>
```

Corresponding query code, to look up an adapter implementing the interfaces:

```
from zope.component import getUtility, getAdapter, getMultiAdapter

# header implements IHeaderBehavior
# doc implements Products.CMFCore.interfaces.IContentish
# request implements zope.publisher.interfaces.browser.IBrowserRequest

from Products.CMFCore.interfaces import IContentish
from zope.publisher.interfaces.browser import IBrowserRequest

self.assertTrue(IHeaderBehavior.providedBy(header))
self.assertTrue(IContentish.providedBy(doc))
self.assertTrue(IBrowserRequest.providedBy(self.portal.REQUEST))

# Throws exception if not found
picker = getMultiAdapter((header, doc, self.portal.REQUEST), IHeaderAnimationPicker)
```

Note: You cannot get adapters on module-level code during import, as the Zope Component Architecture is not yet initialized.

Listing adapter registers

The following code checks whether the IHeaderBehavior adapter is registered correctly:

```
from zope.component import getGlobalSiteManager
sm = getGlobalSiteManager()

registrations = [a for a in sm.registeredAdapters() if a.provided == IHeaderBehavior]
self.assertEqual(len(registrations), 1)
```

Alternative listing adapters

Getting all multi-adapters (context, request):

```
from zope.component import getAdapters
adapters = getAdapters((context, request), provided=Interface)
```

Warning: This does not list locally-registered adapters such as Zope views.

Local adapters

Local adapters are effective only inside a certain container, such as a folder. They use `five.localsitemanager` to register themselves.

- <https://opkode.com/blog/2010/01/26/schema-extending-an-object-only-inside-a-specific-folder/>

Utilities

Description

Utility design pattern in Zope 3 allows overridable singleton class instances for your code.

Introduction

- Utility classes provide site-wide utility functions.
- They are registered by marker interfaces.
- Site customization logic or add-on products can override utilities for enhanced or modified functionality
- Utilities can be looked up by name or interface
- Compared to “plain Python functions”, utilities provide the advantage of being plug-in points without need of *monkey-patching*.

Read more in

- [zope.component documentation](#).

Local and global utilities

Utilities can be

- *global* - registered during Zope start-up
- *local* - registered during add-on installer for a certain site/content item

Local utilities are registered to persistent objects. The context of local utilities is stored in a thread-local variable which is set during traversal. Thus, when you ask for local utilities, they usually come from a persistent registry set up in the Plone site root object.

Global utilities are registered in ZCML and affect all Zope application server and Plone site instances.

Some hints:

```
<Moo^_^> what's difference between gsm.queryUtility() (global site manager) and zope.
↪component.queryUtility()
<agroszer> Moo^_^, I think gsm... takes the global registrations, z.c.queryUtility_
↪respects the current context
```

Registering a global utility

A utility is constructed when Plone is started and ZCML is read. Utilities take no constructor parameters. If you need to use parameters like context or request, consider using views or adapters instead. Utilities may or may not have a name.

- A utility can be provided by a function: the function is called and it returns the utility object.
- A utility can be provided by a class: the class `__call__()` method itself acts as a factory and returns a new class instance.

ZCML example:

```
<!-- Register header animation picking logic - override this for your custom logic -->
<utility
  provides="gomobile.convergence.interfaces.IConvergenceMediaFilter"
  factory=".filter.ConvergedMediaFilter"
/>
```

Python example (named utility):

```
def registerOnsitePaymentProcessor(processor_class):
    """ """

    # Make OnsitePaymentProcessor class available as utiltiy
    processor = processor_class()
    gsm = component.getGlobalSiteManager()
    gsm.registerUtility(processor, interfaces.IOnsitePaymentProcessor, processor.name)
```

The utility class “factory” is in its simplest form a class which implements the interface:

```
class ConvergedMediaFilter(object):
    """ Helper class to deal with media state of content objects.
    """

    zope.interface.implements(IConvergenceMediaFilter)

    def foobar(x):
        """ An example method """
        return x+2
```

Class is constructed / factory is run during the ZCML initialization.

To use this class:

```
from gomobile.convergence.interfaces import IConvergenceMediaFilter

def something():
    filter = getUtility(IConvergenceMediaFilter)
    x = filter.foobar(3)
```

Registering a local utility

- <https://plone.org/documentation/manual/developer-manual/generic-setup/reference/component-registry>
- <http://davisagli.com/blog/registering-add-on-specific-components-using-z3c.baseregistry>
- <https://pypi.python.org/pypi/z3c.baseregistry>

Warning: Local utilities may be destroyed when the add-on product that provides them is reinstalled. Do not use them to store any data.

- <http://markmail.org/thread/twuhyldgyje7p723>

Overriding utility

If you want to override any existing utility you can re-register the utility in the `overrides.zcml` file in your product.

Getting a utility

There are two functions:

`zope.component.getUtility` will raise an exception if the utility is not found.

`zope.component.queryUtility` will return `None` if the utility is not found.

Utility query parameters are passed to the utility class constructor.

Example:

```
from zope.component import getUtility, queryUtility

# context and request are passed to the utility class constructor
# they are optional and depend on the utility itself
picker = getUtility(IHeaderAnimationPicker, context, request)
```

Note: You cannot use `getUtility()` on Python module level code during import, as the Zope Component Architecture is not yet initialized at that time. Always call `getUtility()` from an HTTP request end point or after Zope has been started.

Query local + global utilities:

`zope.component.queryUtility()` for local utilities, with global fallback.

Query only global utilities:

```
from zope.app import zapi
gsm = zapi.getGlobalSiteManager()
return gsm.getUtility(IConvergenceMediaFilter)
```

Warning: Due to Zope component architecture initialization order, you cannot call `getUtility()` in module-level Python code. Module-level Python code is run when the module is being imported, and Zope components are not yet set up at this point.

Getting all named utilities of one interface

Use `zope.component.getUtilitiesFor()`.

Example:

```
def OnsitePaymentProcessors(context):
    """ List all registered on-site payment processors.

    Mostly useful for validating form input.

    Vocabulary contains all payment processors, not just active ones.

    @return: zope.vocabulary.SimpleVocabulary
    """

    utilities = component.getUtilitiesFor(interfaces.IOnsitePaymentProcessor)
    for name, instance in utilities:
        pass
```

Unregistering utilities

- <http://www.muthukadan.net/docs/zca.html#unregisterutility>

Removing persistent local utilities

- *Manually Removing Local Persistent Utilities*
- <http://blog.fourdigits.nl/removing-a-persistent-local-utility>
- <http://blog.fourdigits.nl/removing-a-persistent-local-utility-part-ii>

=== ZCML ===

Description

What Plone programmers should know about ZCML.

Introduction

ZCML stands for the *Zope Configuration Mark-up Language*. It is an XML-based language used to extend and plug into systems based on the Zope Component Architecture (*ZCA*).

It provides:

- conflict resolution (e.g. two plug-ins cannot overlap);
- extensible syntax based on namespaces.

Downsides of ZCML are:

- it is cumbersome to write by hand;
- lack of end-user documentation.

Plone uses ZCML to:

- register components with various places in the system, both core and add-ons.

Note: Everything you can do in ZCML can also be done in Python code.

More info:

- [ZCML reference](#) (does not include Plone specific directives)
- <http://docs.zope.org/zopetoolkit/codingstyle/zcml-style.html>

ZCML workflow

Each Plone component (core, add-on) has a base `configure.zcml` in the package root. This *ZCML* file can include additional nested configuration files using the `<include>` directive.

- ZCML is always interpreted during Plone start-up.
- Your *unit test* may need to manually include ZCML.
- *Funny exception error messages occur if Plone is started in the production mode and ZCML was not properly read for all the packages*

When Plone is started all ZCML files are read.

- New way: Python egg `setup.py` file contains a `autoinclude` hint and is picked up automatically when all the packages are scanned.
- Old way: ZCML reference must be manually added to the `zcml = section` in `buildout.cfg`

If ZCML contains errors *Plone does not start up in the foreground*

Overrides

Besides layer overrides, ZCML provides more hardcore ways to override things in buildout. These overrides can also override utilities etc. and overrides take effect during ZCML parsing, not when site is run.

- Create `overrides.zcml` file in your egg to the same folder as `configure.zcml`
- Syntax is 100% same as in `configure.zcml`
- Restart Plone.

Note: Before Plone 3.3, ZCML directives could not be automatically picked up from eggs. To make Plone pick up the directions in `overrides.zcml`, you'd have to add this line in `buildout.cfg`:

```
zcml =
...
myegg-overrides
```

Since Plone 3.3, the `z3c.autoinclude` plugin can do this (<https://plone.org/products/plone/roadmap/247/>).

Specify files and code from another package

If you ever find yourself needing to use a template from another package, you can do so with using the `configure` tag which will then run the block of *ZCML* in the context of that package.

Here's an example of overriding the *BrowserView* 'folder_contents'. It is defined in package `plone.app.content` in directory `browser` with this *ZCML* statement:

```
<browser:page
    for="Products.CMFCore.interfaces._content.IFolderish"
    class=".folder.FolderContentsView"
    name="folder_contents"
    template="templates/folder_contents.pt"
    permission="cmf.ListFolderContents"
/>
```

In your own package `my.package`, you want to override the class, but keep the template. Assuming you created a class `MyFolderContentsView` inside `foldercontents.py` in the `browser` directory of your package, add this *ZCML* statement:

```
<configure
    xmlns="http://namespaces.zope.org/zope"
    xmlns:browser="http://namespaces.zope.org/browser"
    i18n_domain="my.package">

    <!-- override folder_contents -->
    <configure package="plone.app.content.browser">
        <browser:page
            for="Products.CMFCore.interfaces._content.IFolderish"
            class="my.package.browser.foldercontents.MyFolderContentsView"
            name="folder_contents"
            template="folder_contents.pt"
            layer="my.package.interfaces.IMyPackageLayer"
            permission="cmf.ListFolderContents"
        />
    </configure>
</configure>
```

Basically, you re-define the *BrowserView* in the context of its original package, so that the relative path to the template stays valid. But using the full path in dotted notation, you can let it point to your own class.

Conditionally run ZCML

You can conditionally run *ZCML* if a certain package or feature is installed.

First, include the namespace at the top of the *ZCML* file:

```
<configure
    xmlns="http://namespaces.zope.org/zope"
    xmlns:zcml="http://namespaces.zope.org/zcml"
    i18n_domain="my.package">
    ....
```

Examples

Conditionally run ZCML based upon the installation status of a package:

```
<include zcml:condition="installed some.package" package=".package" />
<include zcml:condition="not-installed some.package" package=".otherpackage" />
```

Conditionally run ZCML based upon the presence of a feature:

```
<include zcml:condition="have plone-4" package=".package" />
<include zcml:condition="not-have plone-4" package=".otherpackage" />
```

Registering features

To register that a feature is present, include the `xmlns:meta` namespace at the top of your *ZCML* file (typically `meta.zcml` in a package), and define a `<meta:provides>` element with your feature's name, like so:

```
<configure
  xmlns="http://namespaces.zope.org/zope"
  xmlns:zcml="http://namespaces.zope.org/zcml"
  xmlns:meta="http://namespaces.zope.org/meta">
  ...
  <meta:provides feature="my-feature-name" />
  ...
</configure>
```

Once registered, you can now use `zcml:condition="have my-feature-name"` to register ZCML configuration that is requires this feature be available.

Add-on Installation And Export framework: GenericSetup

Description

GenericSetup is a framework to modify the Plone site during add-on package installation and uninstallation. It provides XML-based rules to change the site settings.

Introduction

GenericSetup is an XML-based way to import and export Plone site configurations.

It is mainly used to prepare the Plone site for add-on packages, by:

- registering CSS files,
- registering JavaScript files,
- setting various properties,
- registering portlets,
- registering portal_catalog search query indexes,
- ... etc ...

GenericSetup is mostly used to apply an add-on's specific changes to the site configuration and to enable specific behaviors when the add-on installer is run.

GenericSetup XML files are usually in a `profiles/default` folder inside the add-on package.

All run-time through-the-web (TTW) configurable items, like viewlets order through `/@@manage-viewlets` page, are made repeatable using GenericSetup profile files.

You do not need to hand-edit GenericSetup profile files.

You can always change the configuration options through Plone or using the Management Interface, and then you export the resulting profile as an XML file, using the *Export* tab in `portal_setup` accessible from the Management Interface.

Directly editing XML profile files does not change anything on the site, even after Zope restart. This is because run-time TTW configurable items are stored in the database.

If you edit profile files, you need to either reimport the edited files using the `portal_setup` tool or fully rerun the add-on package installer in Plone control panel.

This import will read XML files and change the Plone database accordingly.

Note: Difference between ZCML and GenericSetup

ZCML changes affect loaded Python code in **all** sites inside Zope whereas GenericSetup XML files affect only one Plone site and its database. GenericSetup XML files are always database changes.

Relationship between ZCML and site-specific behavior is usually done using *layers*. ZCML directives, like viewlets and views, are registered to be active only on a certain layer using `layer` attribute. When GenericSetup XML is imported through `portal_setup`, or the add-on package installer is run for a Plone site, the layer is activated for the particular site only, enabling all views registered for this layer.

Note: The `metadata.xml` file (add-on dependency and version information) is read during Plone start-up. If this file has problems, your add-on might not appear in the installer control panel.

- [GenericSetup package page](#).
- [GenericSetup source code](#).

Creating A Profile

You use `<genericsetup>` directive in your add-on package's `configure.zcml`. The name for the default profile executed by the Plone add-on installer is `default`.

If you need different profiles, for example for unit testing, you can declare them here.

XML files for the default profile go in the `profiles/default` folder inside your add-on package.

```
<configure
  xmlns="http://namespaces.zope.org/zope"
  xmlns:genericsetup="http://namespaces.zope.org/genericsetup"
  i18n_domain="your.addonpackage">

  <genericsetup:registerProfile
    name="default"
    title="your.addonpackage (installation profile)"
    directory="profiles/default"
```

```
description='Your add-on package installation profile'
provides="Products.GenericSetup.interfaces.EXTENSION"
/>
```

```
</configure>
```

Multiple Profiles

When you have more than one profile in your add-on package, the add-ons control panel needs to decide which one to use when you install it.

Since Plone 5.1, when there is a `default` profile, it is always used as the installation profile, regardless of other profile names.

Exception: when this `default` profile is marked in an `INonInstallable` utility, it is ignored and Plone falls back to using the first from the alphabetical sorting.

Note: In Plone 5.0 and lower, the profiles are sorted alphabetically by id, and the first one is chosen. If you have profiles `base` and `default`, the `base` profile is installed. It is recommended to let `default` be the alphabetically first profile.

Add-on-specific Issues

Add-on packages may contain:

- A default `GenericSetup` XML profile which is automatically run when the package is installed using the quick-installer. The profile name is usually `default`.
- Other profiles which the user may install using the `portal_setup` *Import* tab, or which can be manually enabled for unit tests.
- An “Import various” step, which runs Python code every time the `GenericSetup` XML profile is installed. See *Custom Installer Code (setuphandlers.py)*.
- A `pre_handler` or `post_handler` when you use `GenericSetup` 1.8.2 or higher. See note at *Custom Installer Code (setuphandlers.py)*.

For more information about custom import steps, see:

- <http://plone.293351.n2.nabble.com/indexing-of-content-created-by-Generic-Setup-td4454703.html>

Listing Available Profiles

Example:

```
# List all profiles know to the Plone instance.
setup_tool = self.portal.portal_setup

profiles = setup_tool.listProfileInfo()
for profile in profiles:
    print str(profile)
```

Sample results:

```
{'product': 'PluggableAuthService',
 'description': 'Content for an empty PAS (plugins registry only).',
 'for': <InterfaceClass Products.PluggableAuthService.interfaces.authservice.
↪IPluggableAuthService>,
 'title': 'Empty PAS Content Profile',
 'version': 'PluggableAuthService-1.5.3',
 'path': 'profiles/empty',
 'type': 1,
 'id': 'PluggableAuthService:empty'}
{'product': 'Products.CMFPlone',
 'description': u'Profile for a default Plone.',
 'for': <InterfaceClass Products.CMFPlone.interfaces.siteroot.IPloneSiteRoot>,
 'title': u'Plone Site',
 'version': u'3.1.7',
 'path': u'/home/moo/sits/parts/plone/CMFPlone/profiles/default',
 'type': 1,
 'id': u'Products.CMFPlone:plone'}
...
```

Installing A Profile

This is usually unit test specific question how to enable certain add-ons for unit testing.

plone.app.testing

See [Product and profile installation](#).

Manually

You might want to install profiles manually if they need to be enabled only for certain tests.

The profile name is in the format `profile-${package_name}:${profile id}`

Unit testing example:

```
# Run the extended profile of the "your.addonpackage" package.
setup_tool.runAllImportStepsFromProfile('profile-your.addonpackage:extended')
```

New in version 4.3.8: Since `Products.GenericSetup 1.8.0` (Plone 4.3.8, Plone 5.0.0) the `profile-` part is optional. The code can handle both.

Missing Upgrade Procedure

In the Add-ons control panel you may see a warning that your add-on package is [missing an upgrade procedure](#).

This means you need to write some *Upgrade steps*.

Uninstall Profile

For the theory, see <http://blog.keul.it/2013/05/how-to-make-your-plone-add-on-products.html>

For an example, see the [collective.pdfpeek](#) source code.

When you deactivate an add-on in the control panel, Plone looks for a profile with the name `uninstall` and applies it.

New in version 5.1: If there is no `uninstall` profile, a warning is displayed before installing the add-on. If you do activate the add-on, no deactivate button will be shown.

New in version 5.1: The add-ons control panel no longer does an automatic partial cleanup, for example removing added skins and css resources. This was always only partial, you could not rely on it to fully cleanup the site.

Note: This method works in Plone 4.3.7 and higher. If you need to support older versions, you may need to write an `Extensions/install.py` file with an `uninstall` method. See older versions of this document for more information.

Dependencies

GenericSetup profile can contain dependencies to other add-on package installers and profiles.

For example, if you want to declare a dependency to the *your.addonpackage* package, that it is automatically installed when your add-on is installed, you can use the declaration below.

This way you can be sure that all layers, portlets and other features which require database changes are usable from *your.addonpackage* when it is run.

`metadata.xml`:

```
<?xml version="1.0"?>
<metadata>
  <version>1000</version>
  <dependencies>
    <dependency>profile-your.addonpackage:default</dependency>
  </dependencies>
</metadata>
```

your.addonpackage declares the profile in its `configure.zcml`:

```
<genericsetup:registerProfile
  name="default"
  title="your.addonpackage"
  directory="profiles/default"
  description='Your add-on package installation profile'
  provides="Products.GenericSetup.interfaces.EXTENSION"
/>
```

Warning: Unlike other GenericSetup XML files, `metadata.xml` is read on the start-up and this read is cached. Always restart Plone after editing `metadata.xml`.

If your `metadata.xml` file contains syntax errors or dependencies to a missing or non-existent package (e.g. due to a typo in a name) your add-on will disappear from the installation control panel.

Note: For some old add-ons in the `Products.*` Python namespace, you must not include the full package name in the dependencies.

This is true when this add-on has registered its profile in Python instead of `zcml`, and there it has used only part of its package name.

In most cases you *do* need to use the full `Products.xxx` name.

To declare a dependency on the simple profile of `Products.PluggableAuthService`:

```
<?xml version="1.0"?>
<metadata>
  <version>1000</version>
  <!-- Install the simple PluggableAuthService profile on the site when this add-on_
  is installed. -->
  <dependencies>
    <dependency>profile-PluggableAuthService:simple</dependency>
  </dependencies>
</metadata>
```

Metadata version numbers

Some old add-on packages may have a `metadata.xml` without version number, but this is considered bad practice.

What should the version number in your `metadata.xml` be?

This mostly matters when you are adding upgrade steps, see also the [Upgrade steps](#) section.

Upgrade steps have a sort order in which they are executed. This used to be alphabetical sorting.

When you had eleven upgrade steps, marked from 1 through 11, alphabetical sorting meant this order: 1, 10, 11, 2, 3, etc.

If you are seeing this, then you are using an old version of `GenericSetup`.

You want numerical sorting here, which is correctly done currently. Versions with dots work fine too.

They get ordered just like they would when used for packages on PyPI.

Best practice for all versions of `GenericSetup` is this:

- Start with 1000. This avoids problems with ancient `GenericSetup` that used alphabetical sorting.
- Simply increase the version by 1 each time you need a new metadata version. For example: 1001, 1002, etc.
- If your add-on package version number changes, but your profile stays the same and no upgrade step is needed, you should **not** change the metadata version. There is no need.
- If you make changes for a new major release, you should increase the metadata version significantly. This leaves room for small metadata version increases on a maintenance branch. Example: You have branch master with version 1025. You make backwards incompatible changes and you increase the version to 2000. You create a maintenance branch where the next metadata version will be 1026.

Custom Installer Code (`setuphandlers.py`)

Besides out-of-the-box XML steps which provide both install and uninstall, `GenericSetup` provides a way to run custom Python code when your add-on package is installed and uninstalled.

This is not a very straightforward process, though.

Note: An easier way may be possible for you. `GenericSetup` 1.8.2 has an option to point to a function to run before or after applying all import steps for your profile.

If you do not need to support older versions, this is the easiest way.

In `configure.zcml`:

```
<configure
  xmlns="http://namespaces.zope.org/zope"
  xmlns:genericsetup="http://namespaces.zope.org/genericsetup"
  i18n_domain="your.addonpackage">

  <genericsetup:registerProfile
    name="default"
    title="Your Add-on Package"
    directory="profiles/default"
    description="A useful package"
    provides="Products.GenericSetup.interfaces.EXTENSION"
    pre_handler="your.addonpackage.setuphandlers.run_before"
    post_handler="your.addonpackage.setuphandlers.run_after"
  />

</configure>
```

In `setuphandlers.py`:

```
def run_before(context):
    # This is run before running the first import step of
    # the default profile. context is portal_setup.
    # If you need the same context as you would get in
    # an import step, like setup_various below, do this:
    profile_id = 'profile-your.addonpackage:default'
    good_old_context = context._getImportContext(profile_id)
    ...

def run_after(context):
    # This is run after running the last import step of
    # the default profile. context is portal_setup.
    ...
```

The best practice is to create a `setuphandlers.py` file which contains a function `setup_various()` which runs the required Python code to make changes to Plone site object.

This function is registered as a custom `genericsetup:importStep` in XML.

Note: When you write a custom `importStep`, remember to write uninstallation code as well.

However, the trick is that all `GenericSetup` import steps, including your custom step, are run for *every* add-on package when they are installed.

If your need to run code which is **specific to your add-on install only** you need to use a marker text file which is checked by the `GenericSetup` context.

Also you need to register this custom import step in `configure.zcml`:

```
<configure
  xmlns="http://namespaces.zope.org/zope"
  xmlns:genericsetup="http://namespaces.zope.org/genericsetup">

  <genericsetup:importStep
    name="your.addonpackage"
    title="your.addonpackage special import handlers"

  />

</configure>
```

```

description=""
handler="your.addonpackage.setuphandlers.setup_various"
/>

```

</configure>

You can run other steps before yours by using the `depends` directive.

For instance, if your import step depends on a content type to be installed first, you must use:

```

<configure
  xmlns="http://namespaces.zope.org/zope"
  xmlns:genericsetup="http://namespaces.zope.org/genericsetup">

  <genericsetup:importStep
    name="your.addonpackage"
    title="your.addonpackage special import handlers"
    description=""
    handler="your.addonpackage.setuphandlers.setup_various">
    <depends name="typeinfo" />
  </genericsetup:importStep>

</configure>

```

setuphandlers.py example

```

__docformat__ = "epytext"

def run_custom_code(site):
    """Run custom add-on package installation code to modify Plone
    site object and others

    @param site: Plone site
    """

def setup_various(context):
    """
    @param context: Products.GenericSetup.context.DirectoryImportContext instance
    """

    # We check from our GenericSetup context whether we are running
    # add-on installation for your package or any other
    if context.readDataFile('your.addonpackage.marker.txt') is None:
        # Not your add-on
        return

    portal = context.getSite()

    run_custom_code(portal)

```

And add a dummy text file `your.addonpackage/your/addonpackage/profiles/default/your.addonpackage.marker.txt`:

This text file can contain any content - it just needs to be present

More information

- <http://keeshink.blogspot.com/2009/02/creating-portal-content-in.html>

- <http://maurits.vanrees.org/weblog/archive/2009/12/catalog> (unrelated, but contains pointers)

Overriding Import Step Order

If you need to override the order of import steps in a package that is not yours, it might work if you [use an overrides.zcml](#).

Controlling The Import Step Execution Order

If you need to control the execution order of one of your own custom import steps, you can do this in your import step definition in zcml.

To make sure the catalog and typeinfo steps are run before your own step, use this code:

```
<configure
  xmlns="http://namespaces.zope.org/zope"
  xmlns:genericsetup="http://namespaces.zope.org/genericsetup"
  i18n_domain="poi">

  <genericsetup:importStep
    name="poi_various"
    title="Poi various import handlers"
    description=""
    handler="Products.Poi.setuphandlers.import_various">
    <depends name="catalog"/>
    <depends name="typeinfo"/>
  </gs:importStep>

</configure>
```

Note: The name that you need, is usually the name of the related xml file, but with the `.xml` stripped. For the `catalog.xml` the import step name is `catalog`. But there are exceptions.

For the `types.xml` and the `types` directory, the import step name is `typeinfo`.

See the section on *Generic Setup files* for a list.

- <http://plone.293351.n2.nabble.com/indexing-of-content-created-by-Generic-Setup-td4454703.html>

Upgrade Steps

You can define upgrade steps to run code when someone upgrades your package from version *x* to *y*.

As an example, let's say that the new version of your `addonpackage` defines a *price* field on a content type *MyType* to be a string, but previously (version 1.1 and earlier) it was a float.

Code that uses this field and assumes it to be a float will break after the upgrade, you'd like to automatically convert existing values for the field to string.

You could do this in a script, but having a `GenericSetup` upgrade step means non-technical people can do it as well.

Once you have the script, it's code can be put in an upgrade step.

Increment Profile Version

First increase the number of the version in the `profiles/default/metadata.xml`. This version number should be an integer.

Package version are different because they add sense like the status of the add-on: is it stable, is it in development, in beta, which branch is it.

A profile version indicates only that you have to migrate data in the database.

Add Upgrade Step

Next we add an upgrade step:

```
<configure
  xmlns="http://namespaces.zope.org/zope"
  xmlns:genericsetup="http://namespaces.zope.org/genericsetup"
  i18n_domain="your.addonpackage">

  <genericsetup:upgradeStep
    title="Convert Price to strings"
    description="Price was previously a float field, it should be converted to_
↪string"
    source="1000"
    destination="1100"
    handler="your.addonpackage.upgrades.convert_price_to_string"
    sortkey="1"
    profile="your.addonpackage:default"
  />

</configure>
```

- You can use a wildcard character for *source* to indicate an upgrade for any previous version. Since Products.GenericSetup 1.7.6 this works fine. To run the upgrade step only when upgrading from a specific version, use that version's number.
- The optional *sortkey* can be used to indicate the order in which upgrade steps from the same source to destination are run.

Add Upgrade Code

The code for the upgrade method itself is best placed in a *upgrades.py* module:

```
from plone import api
import logging

PROFILE_ID = 'profile-your.addonpackage:default'

def convert_price_to_string(context, logger=None):
    """Method to convert float Price fields to string.

    When called from the import_various method, 'context' is
    the plone site and 'logger' is the portal_setup logger.

    But this method will be used as upgrade step, in which case 'context'
```

```
will be portal_setup and 'logger' will be None."""

if logger is None:
    # Called as upgrade step: define our own logger.
    logger = logging.getLogger('your.addonpackage')

# Run the catalog.xml step as that may have defined new metadata
# columns. We could instead add <depends name="catalog"/> to
# the registration of our import step in zcml, but doing it in
# code makes this method usable as upgrade step as well.
# Remove these lines when you have no catalog.xml file.
setup = api.portal.get_tool('portal_setup')
setup.runImportStepFromProfile(PROFILE_ID, 'catalog')

catalog = api.portal.get_tool('portal_catalog')
brains = catalog(portal_type='MyType')
count = 0
for brain in brains:
    current_price = brain.getPrice
    if type(current_price) != type('a string'):
        obj = brain.getObject()
        obj.setPrice(str(current_price))
        obj.reindexObject()
        count += 1

setup.runImportStepFromProfile(PROFILE_ID, 'catalog')
logger.info('%s fields converted.' % count)
```

Other examples of using generic setup to run import steps are below.

If you want to call `types.xml` use `typeinfo`:

```
setup.runImportStepFromProfile(PROFILE_ID, 'typeinfo')
```

If you want to call `workflow.xml` use `workflow`:

```
setup.runImportStepFromProfile(PROFILE_ID, 'workflow')
```

The ids of the various default import steps are defined in several places.

Some of the most used ones are here:

- <https://github.com/zoepfoundation/Products.CMFCore/blob/master/Products/CMFCore/exportimport/configure.zcml>
- <https://github.com/plone/Products.CMFPlone/blob/master/Products/CMFPlone/exportimport/configure.zcml>

After restarting Zope, your upgrade step should be visible in the Management Interface: the `portal_setup` tool has a tab `Upgrades`.

Select your package profile to see which upgrade steps Zope knows about for your add-on.

upgradeDepends

In an upgrade step you can apply a specific import step from your profile:

```
<genericsetup:upgradeDepends
    source="3900"
```

```

destination="4000"
profile="your.addonpackage:default"
title="Apply rolemap.xml.
    This is implicitly taken from the profile this upgradeStep is defined for."
import_steps="rolemap" />

```

You can apply multiple steps, separated by a space:

```

<genericsetup:upgradeDepends
    source="3900"
    destination="4000"
    profile="your.addonpackage:default"
    import_steps="cssregistry jsregistry" />

```

You can apply steps from a different profile:

```

<genericsetup:upgradeDepends
    source="3900"
    destination="4000"
    profile="your.addonpackage:default"
    title="Apply skins.xml from our plone4 profile"
    import_profile="your.addonpackage:plone4"
    import_steps="skins" />

```

You can apply a complete profile:

```

<genericsetup:upgradeDepends
    source="3900"
    destination="4000"
    profile="your.addonpackage:default"
    title="Install our complete cache profile"
    import_profile="your.addonpackage:cache" />

```

Combining Upgrade Steps

You can create many upgrade steps under one migration.

This is useful when you want to have the ability to re-run some parts of the migration and make your code more re-useable (for example cook css resource of your theme).

Here is an example of many upgrade steps you can have to achieve on a site policy:

```

<genericsetup:upgradeSteps
    source="3900"
    destination="4000"
    profile="your.addonpackage:default">

    <genericsetup:upgradeStep
        title="Upgrade addons"
        description="Install and upgrades add-ons"
        handler=".v4.upgrade_addons"
    />

    <genericsetup:upgradeStep
        title="Remove LDAP PAS Plugin"
        description="Execute this upgrade after the plonesite upgrade"
        handler=".v4.upgrade_pas"
    />

```

```
    />

    <genericsetup:upgradeDepends
        title="Apply skins.xml from our plone4 profile"
        import_profile="your.addonpackage:plone4"
        import_steps="skins"
    />

    <genericsetup:upgradeDepends
        title="Install our complete cache profile"
        import_profile="your.addonpackage:cache"
    />

    <genericsetup:upgradeDepends
        title="Apply rolemap.xml.
            This is implicitly taken from the profile the upgradeSteps are defined
↳for: the default"
        import_steps="rolemap"
    />

</genericsetup:upgradeSteps>
```

Add-on Package Appears Twice In The Installer List

This happens if you are developing your own add-on and keep changing things.

You have an error in your add-on package ZCML code which causes portal_quickinstaller to have two entries.

More information

- <http://plone.293351.n2.nabble.com/Product-twice-in-quickinstaller-td5345492.html#a5345492>

Preventing Uninstall

You might want to prevent your add-on package uninstall for some reason.

Example:

```
from zExceptions import BadRequest

def uninstall(self, reinstall):
    if reinstall == False:
        raise BadRequest('This package cannot be uninstalled!')
```

Deprecated since version 5.1: This example is for Extensions/install.py, an old Plone 2 way of writing installers and uninstallers. It is still working in Plone 5.0, but will likely go away in Plone 5.1. In Plone 5.1 you can decide not to create an uninstall profile.

Best Practices

The purge attribute

When importing items such as property sheets, make sure not to override other profile settings: set the `purge` attribute to `False`.

This will *add* the listed items to the property instead of resetting the property.

Example:

```
<property name="metaTypesNotToList" type="lines" purge="False">
  <element value="File"/>
  <element value="Image"/>
</property>
```

The remove Attribute

The `remove` attribute can be used to remove an item.

```
<property name="allowAnonymousViewAbout" type="boolean" remove="true" />
```

There are dangers:

- Some importers do not support the `remove` keyword. They ignore it and add the item blindly. This should be regarded as a bug in the importer. Please report it.
- Some importers check the truth value of the attribute, some just check the presence. `remove="false"` may mean the item stays and may mean it gets removed. Best is to either use `remove="true"` or leave the entire keyword away.

Only Use The Configuration That You Need

When you export your site's configuration, it will include things that you don't need.

For example, if you only need to change the 'Allow anonymous to view about' property, this is what your `propertiestool.xml` should look like:

```
<?xml version="1.0"?>
<object name="portal_properties" meta_type="Plone Properties Tool">
  <object name="site_properties" meta_type="Plone Property Sheet">
    <property name="allowAnonymousViewAbout" type="boolean">True</property>
  </object>
</object>
```

i18n domains in GenericSetup xml files

In your `GenericSetup` profile you can have several xml files. In some of these it makes sense to do translations.

In most of those cases you must use the `plone` domain, but in some you can use your own domain.

Note: You are always allowed to use the `plone` domain, but if the xml file supports a separate domain it is best to use that.

- `actions.xml`: use **your own** domain.

Example:

```
<object name="ducktest" meta_type="CMF Action" i18n:domain="your.addonpackage">
  <property name="title" i18n:translate="">Duck Test</property>
  <property name="description" i18n:translate="">Action: test a duck</property>
  ...
</object>
```

Note: In the portal_actions tool, in the Management Interface, you will see an i18n domain specified for each action.

- catalog.xml: no i18n needed
- componentregistry.xml: no i18n needed
- contenttyperegistry.xml: no i18n needed
- controlpanel.xml: use **your own** domain.

Example:

```
<?xml version="1.0"?>
<object name="portal_controlpanel" meta_type="Plone Control Panel Tool"
  i18n:domain="your.addonpackage" xmlns:i18n="http://xml.zope.org/namespaces/
→i18n">
  <configlet title="Your Add-on Package Configuration Panel" action_id="your.
→addonpackage" appId="your.addonpackage"
    category="Products" condition_expr=""
    icon_expr="string:$portal_url/your_addonpackage_icon.png"
    url_expr="string:${portal_url}/prefs_install_products_form"
    visible="True" i18n:attributes="title">
    <permission>Manage portal</permission>
  </configlet>
</object>
```

- cssregistry.xml: no i18n needed
- diff_tool.xml: no i18n needed
- factorytool.xml: no i18n needed
- jsregistry.xml: no i18n needed
- kssregistry.xml: no i18n needed
- mailhost.xml: no i18n needed
- memberdata_properties.xml: no i18n needed
- metadata.xml: no i18n needed
- portal_atct.xml: use the **plone** domain.

Note: This has no influence on the Collections panel in Site Setup.

It is only used on the edit and criteria pages of an old-style Collection.

- portlets.xml: use the **plone** domain.
- properties.xml: no i18n needed
- propertiestool.xml: no i18n needed

- `rolemap.xml`: no `il8n` needed
- `skins.xml`: no `il8n` needed
- `toolset.xml`: no `il8n` needed
- `types`: use **your own** domain
- `viewlets.xml`: no `il8n` needed
- `workflows`: use the **plone** domain

Generic Setup Files

actionicons.xml

Deprecated since version 4.0.: Plone 5 no longer reads this file, because `Products.CMFActionIcons` is not included. The icons should go in `actions.xml` directly.

actions.xml

Install actions in the `portal_actions` tool.

Example:

```
<?xml version="1.0"?>
<object name="portal_actions" meta_type="Plone Actions Tool"
  xmlns:il8n="http://xml.zope.org/namespaces/il8n">
  <object name="object_buttons" meta_type="CMF Action Category">
    <object name="iterate_checkin" meta_type="CMF Action" il8n:domain="plone">
      <property name="title" il8n:translate="">Check in</property>
      <property name="description" il8n:translate=""></property>
      <property name="url_expr">string:${object_url}/@@content-checkin</property>
      <property name="icon_expr">string:${portal_url}/++resource++checkout.png</
↪property>
      <property name="available_expr">python:path('object/@@iterate_control').checkin_
↪allowed()</property>
      <property name="permissions">
        <element value="View"/>
      </property>
      <property name="visible">True</property>
    </object>
  </object>
</object>
```

These actions are used in various parts of Plone.

These are the object categories in standard Plone:

document_actions Document actions, like rss and print.

site_actions Site actions, like sitemap, accessibility, contact.

object Object tabs, like contents, sharing tab.

object_buttons Object buttons, like delete, rename.

portal_tabs Portal tabs, like Home.

user User actions, like preferences, login, join.

For adding controlpanel actions, see *controlpanel.xml* instead.

The objects support `insert-before` and `insert-after` for inserting the action object before or after another action object.

For removing, use `remove="true"` (or `True`).

Uninstall example:

```
<?xml version="1.0"?>
<object name="portal_actions" meta_type="Plone Actions Tool"
  xmlns:il8n="http://xml.zope.org/namespaces/il8n">
  <object name="object_buttons" meta_type="CMF Action Category">
    <object name="iterate_checkin" remove="true" />
  </object>
</object>
```

Note: You can use your own `il8n` domain.

browserlayer.xml

This registers a specific browser layer, which allows components to be available only when your add-on package is installed.

```
<?xml version="1.0" encoding="UTF-8"?>
<layers>
  <layer name="your.addonpackage"
    interface="your.addonpackage.interfaces.IYourAddonPackageLayer" />
</layers>
```

For removing, use `remove="true"` (or `True`).

Uninstall example:

```
<?xml version="1.0" encoding="UTF-8"?>
<layers>
  <layer name="your.addonpackage" remove="true" />
</layers>
```

componentregistry.xml

Setup items in the local component registry of the Plone Site. The items can be adapters, subscribers or utilities.

This can also be done in `zcml`, which puts it in the global registry that is defined at startup.

The difference is, when you put it in `xml`, the item is only added to a specific Plone Site when you install the package in the add-ons control panel.

Both have their uses.

Example:

```
<?xml version="1.0"?>
<componentregistry>
  <adapters>
    <adapter
```



```

        for="archetypes.multilingual.interfaces.IArchetypesTranslatable"
        provides="plone.app.multilingual.interfaces.ITranslationCloner"
        factory="archetypes.multilingual.cloner.Cloner"
    />
</adapters>
<subscribers>
    <subscriber
        for="archetypes.multilingual.interfaces.IArchetypesTranslatable
            zope.lifecycleevent.interfaces.IObjectModifiedEvent"
        handler="archetypes.multilingual.subscriber.handler"
    />
</subscribers>
<utilities>
    <utility
        interface="Products.ATContentTypes.interface.IATCTTool"
        object="portal_atct"/>
</utilities>
</componentregistry>

```

Note: A subscriber can either have a handler or a factory, not both. A factory must have a provides and may have a name. A subscriber will fail with a provides.

Note: If something does not get added, its provider is probably blacklisted. This list is defined by `Products.GenericSetup.interfaces.IComponentsHandlerBlacklist` utilities. In standard Plone 5, these interfaces are blacklisted as providers:

- `Products.GenericSetup.interfaces.IComponentsHandlerBlacklist`
 - `plone.portlets.interfaces.IPortletManager`
 - `plone.portlets.interfaces.IPortletManagerRenderer`
 - `plone.portlets.interfaces.IPortletType`
-

Uninstall example:

```

<?xml version="1.0"?>
<componentregistry>
    <adapters>
        <adapter
            remove="true"
            for="archetypes.multilingual.interfaces.IArchetypesTranslatable"
            provides="plone.app.multilingual.interfaces.ITranslationCloner"
            factory="archetypes.multilingual.cloner.Cloner"
        />
    </adapters>
    <subscribers>
        <subscriber
            remove="true"
            for="archetypes.multilingual.interfaces.IArchetypesTranslatable
                zope.lifecycleevent.interfaces.IObjectModifiedEvent"
            handler="archetypes.multilingual.subscriber.handler"
        />
    </subscribers>
    <utilities>

```

```
<utility
  remove="true"
  interface="Products.ATContentTypes.interface.IATCTTool"
  object="portal_atct"/>
</utilities>
</componentregistry>
```

Note: The presence of the `remove` keyword is enough. Even if it is empty or contains `false` as value, the item is removed.

Code is in `Products.GenericSetup.components`.

contentrules.xml

Code is in `plone.app.contentrules.exportimport.rules`.

Content Generation

Code is in `Products.GenericSetup.content`:

controlpanel.xml

```
<?xml version="1.0"?>
<object
  name="portal_controlpanel"
  xmlns:i18n="http://xml.zope.org/namespaces/i18n"
  i18n:domain="plone.app.caching">

  <configlet
    title="Caching"
    action_id="plone.app.caching"
    appId="plone.app.caching"
    category="plone-advanced"
    condition_expr=""
    icon_expr="string:${portal_url}/++resource++plone.app.caching.gif"
    url_expr="string:${portal_url}/@@caching-controlpanel"
    visible="True"
    i18n:attributes="title">
      <permission>Manage portal</permission>
    </configlet>

</object>
```

This creates an action in the Site Setup control panel in Plone. Actions are bundled in categories.

The only categories supported in standard Plone 5 are:

- Member (My Preferences)
- Plone (Plone Configuration)
- `plone-advanced` (Advanced)

- plone-content (Content)
- plone-general (General)
- plone-security (Security)
- plone-users (Users)
- Products (Add-on Configuration)

Any other categories are not displayed in the overview control panel. Note that in Plone 4, only Member, Plone and Products were supported. For add-ons, the category Products is recommended.

The `action_id` must be unique over all categories.

Only one permission is allowed.

Uninstall example:

```
<?xml version="1.0"?>
<object name="portal_controlpanel">
  <configlet action_id="plone.app.caching" remove="true" />
</object>
```

Note: The action is removed if the `remove` keyword is `true`. Upper or lower case does not matter.

The action is visible if the `visible` keyword is `true`. Upper or lower case does not matter.

Note: You can use your own i18n domain.

Code is in `Products.CMFPlone.exportimport.controlpanel` and `Products.CMFPlone.PloneControlPanel`.

cssregistry.xml

Deprecated since version 5.0.: This still works in Plone 5, but it is deprecated.

For the old way, see: *Front-end: templates, CSS and JavaScript*.

For the new way, see: `doc:Plone 5 Resource Registry` *</adapt-and-extend/theming/resourceregistry>*.

diff_tool.xml

This is the configuration from `plone.app.contenttypes`:

```
<?xml version="1.0"?>
<object>
  <difftypes>
    <type portal_type="Document">
      <field name="any" difftype="Compound Diff for Dexterity types"/>
    </type>
    <type portal_type="Event">
      <field name="any" difftype="Compound Diff for Dexterity types"/>
    </type>
    <type portal_type="File">
      <field name="any" difftype="Compound Diff for Dexterity types"/>
    </type>
  </difftypes>
</object>
```

```
</type>
<type portal_type="Folder">
  <field name="any" difftype="Compound Diff for Dexterity types"/>
</type>
<type portal_type="Image">
  <field name="any" difftype="Compound Diff for Dexterity types"/>
</type>
<type portal_type="Link">
  <field name="any" difftype="Compound Diff for Dexterity types"/>
</type>
<type portal_type="News Item">
  <field name="any" difftype="Compound Diff for Dexterity types"/>
</type>
</difftypes>
</object>
```

This configures how the difference between two versions of a field are shown on the history tab.

The configuration is stored in the `portal_diff` tool.

For Archetypes content, you need a different `difftype`:

```
<type portal_type="Document">
  <field name="any" difftype="Compound Diff for AT types"/>
</type>
```

A new `difftype` can be registered by calling `Products.CMFDiffTool.CMFDiffTool.registerDiffType`. The `difftypes` in standard Plone 5 are:

- Lines Diff
- Compound Diff for AT types
- Binary Diff
- Field Diff
- List Diff
- HTML Diff
- Compound Diff for Dexterity types

Note: There is no uninstall version. The `remove` keyword is not supported. The `portal_diff` tool does not show configuration for `portal_types` that no longer exist.

Code is in `Products.CMFDiffTool.exportimport.difftool`.

factorytool.xml

This is only needed for Archetypes content types. It makes sure when you start adding a content item but don't finish it, that no half created item lingers.

This is not needed for Dexterity items.

```
<?xml version="1.0"?>
<object name="portal_factory" meta_type="Plone Factory Tool">
  <factorytypes>
```

```

<type portal_type="Document"/>
<type portal_type="Event"/>
<type portal_type="File"/>
<type portal_type="Folder"/>
<type portal_type="Image"/>
<type portal_type="Link"/>
<type portal_type="News Item"/>
<type portal_type="Topic"/>
</factorytypes>
</object>

```

Note: The `remove` keyword is not supported.

Code is in `Products.ATContentTypes.exportimport.factorytool`.

jsregistry.xml

Deprecated since version 5.0.: This still works in Plone 5, but it is deprecated.

For the old way, see: *Front-end: templates, CSS and JavaScript*.

For the new way, see: `doc:Plone 5 Resource Registry` `</adapt-and-extend/theming/resourceregistry>`.

kssregistry.xml

Deprecated since version 4.3: Instead of these Kinetic Style Sheets you should use javascript.

metadata.xml

The `metadata.xml` file is read during Plone start-up.

If this file has problems your add-on package might not appear in the installer control panel.

The `metadata.xml` file contains add-on dependency and version information.

```

<?xml version="1.0"?>
<metadata>
  <version>1000</version>
  <dependencies>
    <dependency>profile-your.addonpackage:default</dependency>
  </dependencies>
</metadata>

```

The dependencies are optional.

There is no import step that reads this file. The `portal_setup` tool uses this information when installing a profile.

It installs the profiles that are listed as dependencies, before installing your own profile.

Since `Products.GenericSetup 1.8.0`, dependency profiles that are already installed, are not installed again.

Instead, their upgrade steps, are applied, if they have them.

After your profile is installed, `portal_setup` stores the version number. This is used when determining if any upgrade steps are available for your profile.

When you search for `metadata.xml` in the documentation, you will find more information in context.

Note: There is no uninstall version of `metadata.xml`. An uninstall profile can have its own `metadata.xml` with a version and even profiles. But for dependencies no purge or remove keyword is supported.

portal_atct.xml

Deprecated since version 4.2: This was used for Archetypes Topics, the old-style Collections. Since Plone 4.2 you can use new-style Collections. Please use those.

```
<?xml version="1.0"?>
<atcttool
  xmlns:i18n="http://xml.zope.org/namespaces/i18n">
  <topic_indexes i18n:domain="plone">
    <index name="allowedRolesAndUsers"
      description="The roles and users with View permission on an item"
      enabled="False" friendlyName="Internal Security"
      i18n:attributes="description; friendlyName" />
    <index name="created" description="The time and date an item was created"
      enabled="True" friendlyName="Creation Date"
      i18n:attributes="description; friendlyName">
      <criteria>ATFriendlyDateCriteria</criteria>
      <criteria>ATDateRangeCriterion</criteria>
    </index>
  </topic_indexes>
  <topic_metadata i18n:domain="plone">
    <metadata name="created"
      description="The time and date an item was created"
      enabled="False" friendlyName="Creation Date"
      i18n:attributes="description; friendlyName" />
  </topic_metadata>
  <property name="title">ATContentTypes Tool</property>
  <property name="image_types">
    <element value="Image" />
    <element value="News Item" />
  </property>
  <property name="folder_types">
    <element value="Image" />
  </property>
  <property name="album_batch_size">30</property>
  <property name="album_image_scale">thumb</property>
  <property name="single_image_scale">preview</property>
</atcttool>
```

portal_placeful_workflow

This handles the `portal_placeful_workflow.xml` file and the `portal_placeful_workflow` directory.

This install or configures a placeful workflow.

For this to work, you must install Workflow Policy Support (CMFPlacefulWorkflow) in the Add-ons control panel. This package is included in standard Plone, but does not come installed by default.

Standard `portal_placeful_workflow.xml` from `Products.CMFPlacefulWorkflow`:

```
<?xml version="1.0"?>
<object name="portal_placeful_workflow" meta_type="Placeful Workflow Tool">
  <property name="title"></property>
  <property name="max_chain_length" type="int">1</property>
  <object name="intranet" meta_type="WorkflowPolicy"/>
  <object name="old-plone" meta_type="WorkflowPolicy"/>
  <object name="one-state" meta_type="WorkflowPolicy"/>
  <object name="simple-publication" meta_type="WorkflowPolicy"/>
</object>
```

Standard portal_placeful_workflow/simple-publication.xml from Products.CMFPlacefulWorkflow:

```
<?xml version="1.0"?>
<object name="simple-publication" meta_type="WorkflowPolicy">
  <property name="title">Simple publication</property>
  <bindings>
    <default>
      <bound-workflow workflow_id="simple_publication_workflow"/>
    </default>
    <type default_chain="true" type_id="Document"/>
    <type default_chain="true" type_id="Event"/>
    <type type_id="File">
    </type>
    <type default_chain="true" type_id="Folder"/>
    <type type_id="Image">
    </type>
    <type default_chain="true" type_id="Link"/>
    <type default_chain="true" type_id="News Item"/>
    <type default_chain="true" type_id="Collection"/>
  </bindings>
</object>
```

Uninstall example:

```
<?xml version="1.0"?>
<object name="portal_placeful_workflow" meta_type="Placeful Workflow Tool">
  <object name="old-plone" meta_type="WorkflowPolicy" remove="true" />
</object>
```

The import handler is in Products.CMFPlacefulWorkflow.exportimport.importWorkflowPolicies.

portlets.xml

Code is in plone.app.portlets.exportimport.portlets.

propertyiestool.xml

Deprecated since version 5.0: Most properties are now handled in the configuration registry and can be configured in registry.xml.

In propertyiestool.xml you can change all values of the portal_properties tool. Example:

```
<?xml version="1.0"?>
<object name="portal_properties" meta_type="Plone Properties Tool">
  <object name="site_properties" meta_type="Plone Property Sheet">
    <property name="use_email_as_login" type="boolean">False</property>
    <property name="default_editor" type="string">TinyMCE</property>
  </object>
</object>
```

Uninstall example:

```
<?xml version="1.0" encoding="UTF-8"?>
<object name="portal_properties" meta_type="Plone Properties Tool">
  <object name="site_properties" meta_type="Plone Property Sheet">
    <property name="default_page_types" type="lines" purge="False">
      <element value="UninstallTest" remove="True" />
    </property>
  </object>
</object>
```

pluginregistry.xml

This configures PAS plugin orderings and active plugins. It isn't part of Plone itself, it is used by other frameworks and can be used in Plone with a little extra configuration.

First, you need a monkey patch in your `__init__.py` to point the importer at where Plone keeps its PAS plugins.

```
from Products.PluginRegistry import exportimport
from Products.PluginRegistry.interfaces import IPluginRegistry

def getRegistry(site):
    return IPluginRegistry(site.acl_users.plugins)

exportimport._getRegistry = getRegistry
```

Secondly, code to handle the import step needs to be activated in Plone:

```
<genericsetup:importStep
  name="PAS Plugin Registry"
  title="PAS Plugin Registry"
  description=""
  handler="Products.PluginRegistry.exportimport.importPluginRegistry"
/>
```

Now you can use `pluginregistry.xml` in your generic setup profiles:

```
<?xml version="1.0"?>
<plugin-registry>
  <plugin-type id="IAAuthenticationPlugin"
    title="authentication"
    description="Authentication plugins are responsible for validating_
    ↪ credentials generated by the Extraction Plugin."
    interface="Products.PluggableAuthService.interfaces.plugins.
    ↪ IAAuthenticationPlugin">
    <plugin id="source_users"/>
    <plugin id="session"/>
  </plugin-type>
</plugin-registry>
```



```

    <plugin id="sql"/>
  </plugin-type>

  <plugin-type id="IPropertiesPlugin" title="properties"
    description="Properties plugins generate property sheets for users."
    interface="Products.PluggableAuthService.interfaces.plugins.
↳IPropertiesPlugin">
    <plugin id="sql" />
    <plugin id="mutable_properties"/>
  </plugin-type>

  <plugin-type id="IRolesPlugin" title="roles"
    description="Roles plugins determine the global roles which a user has."
    interface="Products.PluggableAuthService.interfaces.plugins.IRolesPlugin">
    <plugin id="portal_role_manager"/>
    <plugin id="sql"/>
  </plugin-type>

  <plugin-type id="IUserEnumerationPlugin"
    title="user_enumeration"
    description="Enumeration plugins allow querying users by ID, and
↳searching for users who match particular criteria."
    interface="Products.PluggableAuthService.interfaces.plugins.
↳IUserEnumerationPlugin">
    <plugin id="source_users"/>
    <plugin id="mutable_properties"/>
    <plugin id="sql"/>
  </plugin-type>

  <plugin-type id="IUserAdderPlugin" title="user_adder"
    description="User Adder plugins allow the Pluggable Auth Service to
↳create users."
    interface="Products.PluggableAuthService.interfaces.plugins.
↳IUserAdderPlugin">
    </plugin-type>
</plugin-registry>

```

registry.xml

This edits the configuration registry.

Note: The name of this import step is `plone.app.registry`, **not** `registry`.

Example for adding all records of an interface:

```

<?xml version="1.0"?>
<registry>
  <records interface="plone.app.iterate.interfaces.IIterateSettings" />
</registry>

```

Example for adding an individual record:

```

<?xml version="1.0"?>
<registry>

```

```
<record name="your.addonpackage.timeout">
  <field type="plone.registry.field.Int">
    <title>Timeout</title>
    <min>0</min>
  </field>
  <value>100</value>
</record>
</registry>
```

Uninstall example:

```
<?xml version="1.0"?>
<registry>
  <records interface="plone.app.iterate.interfaces.IIterateSettings" remove="true" />
  <record name="your.addonpackage.timeout" remove="true" />
</registry>
```

The item is removed if the `remove` keyword is `true`. Upper or lower case does not matter.

Existing values of lists are purged by default. The values are not purged if the `purge` keyword is `false`.

Upper or lower case does not matter.

For more examples, see the [plone.app.registry](#) documentation.

Code is in `plone.app.registry.exportimport.handler`.

repositorytool.xml

This handles the versioning policy of content.

The default configuration in Plone is:

```
<?xml version="1.0"?>
<repositorytool>
  <policies purge="true">
    <policy name="at_edit_autoversion"
      title="Create version on edit (AT objects only)"
      class="Products.CMFEditions.VersionPolicies.ATVersionOnEditPolicy"/>
    <policy name="version_on_revert" title="Create version on version revert"/>
  </policies>
  <polycymap purge="true">
    <type name="ATDocument">
      <policy name="at_edit_autoversion"/>
      <policy name="version_on_revert"/>
    </type>
    <type name="ATNewsItem">
      <policy name="at_edit_autoversion"/>
      <policy name="version_on_revert"/>
    </type>
    <type name="Document">
      <policy name="at_edit_autoversion"/>
      <policy name="version_on_revert"/>
    </type>
    <type name="Event">
      <policy name="at_edit_autoversion"/>
      <policy name="version_on_revert"/>
    </type>
  </polycymap>
</repositorytool>
```

```

<type name="Link">
  <policy name="at_edit_autoversion"/>
  <policy name="version_on_revert"/>
</type>
<type name="News Item">
  <policy name="at_edit_autoversion"/>
  <policy name="version_on_revert"/>
</type>
</policymap>
</repositorytool>

```

Code is in `Products.CMFEditions.exportimport.repository`.

rolemap.xml

In `rolemap.xml` you define new roles and grant permissions. Both are optional.

```

<?xml version="1.0"?>
<rolemap>
  <roles>
    <role name="Anonymous"/>
    <role name="Authenticated"/>
    <role name="Manager"/>
    <role name="Site Administrator"/>
    <role name="Member"/>
    <role name="Owner"/>
    <role name="Reviewer"/>
    <role name="Reader" />
    <role name="Editor" />
    <role name="Contributor" />
  </roles>
  <permissions>
    <permission name="Pass the bridge"
      acquire="True">
      <role name="Manager"/>
      <role name="Site Administrator"/>
    </permission>
  </permissions>
</rolemap>

```

The roles above are the standard roles in Plone 5. In your profile you only need to list other roles.

The permission must already exist on the Zope level, otherwise you get an error when installing your profile:

```
ValueError: The permission <em>Pass the bridge</em> is invalid.
```

A permission is created on the Zope level when it is used in code. See [Creating permissions](#).

When a role in a permission does not exist, it is silently ignored. The roles listed in a permission are not added.

They replace all existing roles.

With `acquire="true"` (or `True`, `yes`, `1`) roles are also acquired from the Zope root.

Note: There is no `uninstall` version for `rolemap.xml`. `purge` and `remove` are not supported. You can set different values for a permission if this makes sense in your case. This will reset the permission to the same settings as on the Zope level:

```
<permission name="Pass the bridge" acquire="True" />
```

sharing.xml

The sharing.xml file let you add custom roles to the sharing tab. For reference, visit: [Local Roles](#).

skins.xml

Skins are old fashioned, you may not need this. The more modern way is: use browser views and static directories. But skins are still installed by several add-on packages.

Example:

```
<?xml version="1.0"?>
<object name="portal_skins" meta_type="Plone Skins Tool">
  <object name="ATContentTypes" meta_type="Filesystem Directory View"
    directory="Products.ATContentTypes:skins/ATContentTypes"/>
  <skin-path name="*">
    <layer name="ATContentTypes" insert-after="custom"/>
  </skin-path>
</object>
```

- The object is added to the Contents tab of portal_skins.
- The layer is added to one or more skin selections on the Properties tab of portal_skins.
- The skin-path name is usually * to add the skin layer to all skin selections (old style themes in portal_skins). It can also contain a specific skin, for example Plone Default, Sunburst Theme, Plone Classic Theme.

You can set a few properties on the portal_skins object. Products.CMFPlone sets good defaults which you should keep:

```
<?xml version="1.0"?>
<object name="portal_skins"
  meta_type="Plone Skins Tool"
  default_skin="Plone Default"
  allow_any="False"
  cookie_persistence="False"
  request_varname="plone_skin">
</object>
```

- The meta_type should always be Plone Skins Tool or be removed. It is ignored.
- default_skin is the name of the default skin selection.
- allow_any indicates whether users are allowed to use arbitrary skin paths.
- cookie_persistence indicates whether the skins cookie is persistent or not.
- request_varname gets the variable name to look for in the request.

The allow_any, cookie_persistence and request_varname options are old functionality and seem not well supported anymore. No cookie is set. You can choose a skin path even when allow_any is false.

The idea is: if you have the Sunburst Theme as default, and also have the Plone Classic Theme available, you can view the site in the classic theme by visiting this link: http://localhost:8080/Plone?plone_skin=Plone%20Classic%20Theme

Uninstall example:

```
<?xml version="1.0"?>
<object name="portal_skins" meta_type="Plone Skins Tool">
  <object name="ATContentTypes" remove="true" />
  <skin-path name="*">
    <layer name="ATContentTypes" remove="true" />
  </skin-path>
</object>
```

Note: For removing the layer `remove=""` is sufficient. For removing the object `remove="true"` is required. Recommended is to use the full `remove="true"` in both cases.

tinymce.xml

Deprecated since version 5.0: Since Plone 5.0 this is done in `registry.xml` instead. The fields are defined in `Products.CMFPlone.interfaces.controlpanel`.

Import TinyMCE settings.

Code is in `Products.TinyMCE.exportimport`.

toolset.xml

This is used to add a tool to the site.

Warning: This is an old way and should not be used in new code. You should probably register a utility instead of a tool. `componentregistry.xml` might be an alternative, but registering a utility in `zcml` would be better. If the utility needs configuration, you can use `registry.xml`.

Example:

```
<?xml version="1.0"?>
<tool-setup>
  <required tool_id="portal_atct"
    class="Products.ATContentTypes.tool.atct.ATCTTool"/>
  <required tool_id="portal_factory"
    class="Products.ATContentTypes.tool.factory.FactoryTool"/>
  <required tool_id="portal_metadata"
    class="Products.ATContentTypes.tool.metadata.MetadataTool"/>
</tool-setup>
```

Uninstall example:

```
<?xml version="1.0"?>
<tool-setup>
  <forbidden tool_id="portal_atct" />
  <forbidden tool_id="portal_factory" />
  <forbidden tool_id="portal_metadata" />
</tool-setup>
```

Note: Adding a forbidden tool that was first required, like in the example above, is only supported since Products.GenericSetup 1.8.3.

Code is in:

- Products.GenericSetup.registry._ToolsetParser
- Products.GenericSetup.registry.ToolsetRegistry
- Products.GenericSetup.tool.importToolset

typeinfo

This handles the `types.xml` file and the `types` directory.

Note: The name of this import step is `typeinfo`, **not** `types`.

Partial example from `plone.app.contenttypes`:

```
<?xml version="1.0"?>
<object meta_type="Plone Types Tool" name="portal_types">
  <object meta_type="Dexterity FTI" name="Collection" />
  <object meta_type="Dexterity FTI" name="Document" />
  <object meta_type="Dexterity FTI" name="Folder" />
  <object meta_type="Dexterity FTI" name="Link" />
  <object meta_type="Dexterity FTI" name="File" />
  <object meta_type="Dexterity FTI" name="Image" />
  <object meta_type="Dexterity FTI" name="News Item" />
  <object meta_type="Dexterity FTI" name="Event" />
  <object name="Plone Site"
    meta_type="Factory-based Type Information with dynamic views"/>
</object>
```

This adds content types in the `portal_types` tool. The `meta_type` can be:

- `Dexterity FTI` for `Dexterity` content. This is probably what you want.
- `Factory-based Type Information with dynamic views` for `Archetypes` content and for the `Plone Site` itself
- `Factory-based Type Information` for `Archetypes` content that does not need dynamic views, the ability to choose a view in the display menu.

The `types.xml` should be accompanied by a `types` folder with details information on the new types. If you are editing an already existing type, then `types.xml` is not needed: a file in the `types` folder is enough.

If the object name in `types.xml` is `Collection` then you must add a file `types/Collection.xml`. This file is in `plone.app.contenttypes`:

```
<?xml version="1.0"?>
<object name="Collection"
  meta_type="Dexterity FTI"
  i18n:domain="plone" xmlns:i18n="http://xml.zope.org/namespaces/i18n">

  <!-- Basic metadata -->
  <property name="title" i18n:translate="">Collection</property>
```

```

<property name="description"
  i18n:translate="">Collection</property>
<property name="global_allow">True</property>
<property name="filter_content_types">False</property>
<property name="allowed_content_types" />
<property name="allow_discussion">False</property>
<property name="add_permission">plone.app.contenttypes.addCollection</property>
<property name="klass">plone.app.contenttypes.content.Collection</property>
<property name="schema"></property>
<property name="model_source"></property>
<property name="model_file">plone.app.contenttypes.schema:collection.xml</property>
<property name="behaviors" purge="false">
  <element value="plone.app.content.interfaces.INameFromTitle"/>
  <element value="plone.app.contenttypes.behaviors.collection.ICollection"/>
  <element value="plone.app.dexterity.behaviors.discussion.IAllowDiscussion"/>
  <element value="plone.app.dexterity.behaviors.id.IShortName"/>
  <element value="plone.app.dexterity.behaviors.exclfromnav.IExcludeFromNavigation"/
→>
  <element value="plone.app.dexterity.behaviors.metadata.IDublinCore"/>
  <element value="plone.app.contenttypes.behaviors.richtext.IRichText"/>
  <element value="plone.app.relationfield.behavior.IRelatedItems"/>
  <element value="plone.app.lockingbehavior.behaviors.ILocking" />
</property>

<!-- View information -->
<property name="default_view">listing_view</property>
<property name="default_view_fallback">False</property>
<property name="view_methods">
  <element value="listing_view"/>
  <element value="summary_view"/>
  <element value="tabular_view"/>
  <element value="full_view"/>
  <element value="album_view"/>
  <element value="event_listing"/>
</property>

<!-- Method aliases -->
<alias from="(Default)" to="(dynamic view)"/>
<alias from="edit" to="@@edit"/>
<alias from="sharing" to="@@sharing"/>
<alias from="view" to="(selected layout)" />

<!-- Actions -->
<action title="View" action_id="view" category="object" condition_expr=""
  url_expr="string:${object_url}" visible="True">
  <permission value="View"/>
</action>

<action title="Edit" action_id="edit" category="object" condition_expr=""
  url_expr="string:${object_url}/edit" visible="True">
  <permission value="Modify portal content"/>
</action>
</object>

```

For comparison, here is the `types.xml` from `plone.app.collection` which has an old style Archetypes Collection:

```
<?xml version="1.0"?>
<object name="portal_types">
  <!-- We remove the existing FTI since it could be Dexterity-based and would
       not be compatible in that case.  You get this error when installing:
       ValueError: undefined property 'content_meta_type' -->
  <object name="Collection" remove="True"/>
  <object name="Collection"
    meta_type="Factory-based Type Information with dynamic views" />
</object>
```

And here is the types/Collection.xml from plone.app.collection:

```
<?xml version="1.0"?>
<object name="Collection"
  meta_type="Factory-based Type Information with dynamic views"
  i18n:domain="plone" xmlns:i18n="http://xml.zope.org/namespaces/i18n">
  <property name="title" i18n:translate="">Collection</property>
  <property name="description"
    i18n:translate="">Collection of searchable information</property>
  <property name="icon_expr"></property>
  <property name="content_meta_type">Collection</property>
  <property name="product">plone.app.collection</property>
  <property name="factory">addCollection</property>
  <property name="immediate_view">standard_view</property>
  <property name="global_allow">True</property>
  <property name="filter_content_types">True</property>
  <property name="allowed_content_types"/>
  <property name="allow_discussion">False</property>
  <property name="default_view">standard_view</property>
  <property name="view_methods">
    <element value="standard_view" />
    <element value="summary_view" />
    <element value="all_content" />
    <element value="tabular_view" />
    <element value="thumbnail_view" />
  </property>
  <alias from="(Default)" to="(dynamic view)" />
  <alias from="edit" to="atct_edit" />
  <alias from="sharing" to="@@sharing" />
  <alias from="view" to="(selected layout)" />
  <action title="View" action_id="view" category="object" condition_expr=""
    url_expr="string:${object_url}/" visible="True">
    <permission value="View" />
  </action>
  <action title="Edit" action_id="edit" category="object" condition_expr=""
    url_expr="string:${object_url}/edit" visible="True">
    <permission value="Modify portal content" />
  </action>
</object>
```

Uninstall example:

```
<?xml version="1.0"?>
<object name="portal_types">
  <object name="Collection" remove="true"/>
</object>
```

Note: The `remove` keyword is supported for actions. `remove=""` is enough, but recommended is to use `remove="true"`.

The `view_methods` property is a list that is always imported fresh. Elements that are not in the list, are removed. If you only want to add an element and want to keep any existing elements, you can tell it not to purge:

```
<property name="view_methods" purge="False">
  <element value="new_view" />
</property>
```

This does not work for the `allowed_content_types`: they are always purged.

Note: You can use your own i18n domain.

viewlets.xml

workflows.xml

This handles the `workflows.xml` file and the `workflows` directory.

Example from `Products/CMFPlone/profiles/default/workflows.xml` in Plone 5.0:

```
<?xml version="1.0"?>
<object name="portal_workflow" meta_type="Plone Workflow Tool">
  <property
    name="title">Contains workflow definitions for your portal</property>
  <object name="folder_workflow" meta_type="Workflow"/>
  <object name="intranet_folder_workflow" meta_type="Workflow"/>
  <object name="intranet_workflow" meta_type="Workflow"/>
  <object name="one_state_workflow" meta_type="Workflow"/>
  <object name="plone_workflow" meta_type="Workflow"/>
  <object name="simple_publication_workflow" meta_type="Workflow"/>
  <bindings>
    <default>
      <bound-workflow workflow_id="simple_publication_workflow"/>
    </default>
    <type type_id="ATBooleanCriterion"/>
    <type type_id="ATCurrentAuthorCriterion"/>
    <type type_id="ATDateCriteria"/>
    <type type_id="ATDateRangeCriterion"/>
    <type type_id="ATListCriterion"/>
    <type type_id="ATPathCriterion"/>
    <type type_id="ATPortalTypeCriterion"/>
    <type type_id="ATReferenceCriterion"/>
    <type type_id="ATRelativePathCriterion"/>
    <type type_id="ATSelectionCriterion"/>
    <type type_id="ATSimpleIntCriterion"/>
    <type type_id="ATSimpleStringCriterion"/>
    <type type_id="ATSortCriterion"/>
    <type type_id="Discussion Item"/>
    <type type_id="File"/>
    <type type_id="Image"/>
    <type type_id="Plone Site"/>
  </bindings>
</object>
```

```
</bindings>
</object>
```

This adds six workflows in the `portal_workflow` tool. It sets the default workflow to `simple_publication_workflow`.

It sets several types to not use any workflow.

Next to this, the `workflows` directory is checked. This contains sub directories with the same name as the workflows. Each sub directory contains a file `definition.xml` with the definition for this workflow.

See [the Plone workflows](#).

Code is in `Products.DCWorkflow.exportimport`.

Events

Description

How to add event hooks to your Plone code to perform actions when something happens on a Plone site.

Introduction

This document briefly discusses event handling using the `zope.event` module. The Zope Component Architecture's [zope.event package](#) is used to manage subscribable events in Plone.

Some of the notable characteristics of the Plone event system are:

- it is simple;
- subscriber calling order is not modifiable — you cannot set the order in which event handlers are called;
- events cannot be cancelled — all handlers will always get the event;
- event handlers cannot have return values;
- exceptions raised in an event handler will interrupt the request processing.

Registering an event handler

Plone events can be scoped:

- *globally* (no scope)
- *per content type*

Example: Register an event-handler on your contenttype's creation

In `your.product/your/product/configure.zcml` insert:

```
<subscriber
  for=".interfaces.IMyContentTypeClass
    zope.lifecycleevent.IObjectCreatedEvent"
  handler=".your_python_file.your_method"
/>
```

The first line defines to which interface you want to bind the execution of your code, which means here, that the code will only be executed if the object is one of your contenttype's. If you want this to be interface-agnostic, insert an asterix as a wildcard instead.

The second line defines the event on which this should happen, which is here 'IOBJECTCREATEDEvent' – for Archetypes you should use 'Products.Archetypes.interfaces.IObjectInitializedEvent' instead. For more available possible events to be used as a trigger, see event handler documentation

The third line gives the path to the script that is supposed to be executed.

Create your.product/your/product/your_python_file.py and insert:

```
def your_method(object, event):

    # do sth with your created contenttype
```

For Dexterity-contenttype's and additional ZOPE-Illumination see also: event handler documentation

Subscribing using ZCML

Subscribing to a global event using *ZCML*.

```
<subscriber
  for="Products.PlonePAS.events.UserLoggedOutEvent"
  handler=".smartcard.clear_extra_cookies_on_logout"
/>
```

For this event, the Python code in `smartcard.py` would be:

```
def clear_extra_cookies_on_logout(event):
    # What event contains depends on the
    # triggerer of the event and event class
    request = event.object.REQUEST
    ...
```

Custom event example subscribing to all `IMyEvents` when fired by `IMyObject`:

```
<subscriber
  for=".interfaces.IMyObject
    .interfaces.IMyEvent"
  handler=".content.MyObject.myEventHandler"
/>
```

Life cycle events example:

```
<subscriber
  zcml:condition="installed zope.lifecycleevent"
  for=".interfaces.ISitsPatient
    zope.lifecycleevent.IObjectModifiedEvent"
  handler=".content.SitsPatient.objectModified"
/>
```

Subscribing using Python

The following subscription is valid through the process life cycle. In unit tests, it is important to clear test event handlers between the test steps.

Example:

```
import zope.component

def my_event_handler(context, event):
    """
    @param context: Zope object for which the event was fired. Usually this is a
    ↪Plone content object.

    @param event: Subclass of event.
    """
    pass

gsm = zope.component.getGlobalSiteManager()
gsm.registerHandler(my_event_handler, (IMyObject, IMyEvent))
```

Firing an event

Use `zope.event.notify()` to fire event objects to their subscribers.

Example of how to fire an event in unit tests:

```
import zope.event
from plone.postpublicationhook.event import AfterPublicationEvent

event = AfterPublicationEvent(self.portal, self.portal.REQUEST)
zope.event.notify(event)
```

Event types

Creation events

Products.Archetypes.interfaces.IObjectInitializedEvent is fired for an Archetypes-based object when it's being initialised; i.e. when it's being populated for the first time.

Products.Archetypes.interfaces.IWebDAVObjectInitializedEvent is fired for an Archetypes-based object when it's being initialised via WebDAV.

zope.lifecycleevent.IObjectCreatedEvent is fired for all Zopeish objects when they are being created (they don't necessarily need to be content objects).

Warning: Archetypes and Zope 3 events might not be compatible with each other. Please see links below.

Other resources:

- <https://plone.org/documentation/manual/developer-manual/archetypes/other-useful-archetypes-features/how-to-use-events-to-hook-the-archetypes-creation-process>
- <http://n2.nabble.com/IObjectInitializedEvent-tp4784897p4784897.html>

Modified events

Two different content event types are available and might work differently depending on your scenario:

Products.Archetypes.interfaces.IObjectEditedEvent called for Archetypes-based objects that are not in the creation stage any more.

Note: `Products.Archetypes.interfaces.IObjectEditedEvent` is fired after `reindexObject()` is called. If you manipulate your content object in a handler for this event, you need to manually reindex new values, or the changes will not be reflected in the `portal_catalog`.

zope.lifecycleevent.IObjectModifiedEvent called for creation-stage events as well, unlike the previous event type.

Products.Archetypes.interfaces.IWebDAVObjectEditedEvent called for Archetypes-based objects when they are being edited via WebDAV.

Products.Archetypes.interfaces.IEditBegunEvent called for Archetypes-based objects when an edit operation is begun.

Products.Archetypes.interfaces.IEditCancelledEvent called for Archetypes-based objects when an edit operation is canceled.

Delete events

Delete events can be fired several times for the same object. Some delete event transactions are rolled back.

- Read more about Delete events in [this discussion](#).

Copy events

zope.lifecycleevent.IObjectCopiedEvent is triggered when an object is copied.

Workflow events

Products.DCWorkflow.interfaces.IBeforeTransitionEvent is triggered before a workflow transition is executed.

Products.DCWorkflow.interfaces.IAfterTransitionEvent is triggered after a workflow transition has been executed.

The DCWorkflow events are low-level events that can tell you a lot about the previous and current states.

Products.CMFCore.interfaces.IActionSucceededEvent this is a higher level event that is more commonly used to react after a workflow action has completed.

Zope startup events

zope.processlifetime.IProcessStarting is triggered after component registry has been loaded and Zope is starting up.

zope.processlifetime.IDatabaseOpened is triggered after the main ZODB database has been opened.

Asynchronous event handling

- <http://stackoverflow.com/questions/15875088/running-plone-subscriber-events-asynchronously>

See also

- <https://pypi.python.org/pypi/zope.event/3.4.1>
- http://apidoc.zope.org/++apidoc++/ZCML/http_co__sl__sl_namespaces.zope.org_sl_zope/subscriber/index.html
- `zope.component.registry`

Customizing Plone

Introduction

Plone can be customized in two different ways, depending on which kind of component you are trying to change:

- Through-the-web.
- By add-on products.

You should never edit files directly in an egg folder. Instead you usually create a customized version of the item you wish to modify and then configure Plone to use your customized version instead of the stock one.

Through-the-web changes

Minor configuration changes can be done through the web. These changes are effective immediately and don't require you to write any code or restart Zope application server. The downside is that since through-the-web changes don't have a source code "recipe" for what you did, the changes are not automatically repeatable. If you need to do the same changes for another site again, or you need heavily modify your site, you need go through manual steps to achieve the same customization.

Possible through-the-web changes are:

- Site settings: E.g. adding/removing *content rules*
- Showing and hiding viewlets (parts of the page) using `@@manage-viewlets`
- Exporting and importing parts of the site configuration in `portal_setup`
- Customizing viewlet templates in `portal_view_customization`
- Customize `portal_skins` layer theme files in `portal_skins`
- Uploading JavaScript files, CSS files and images through Zope management interface and registering using `portal_css` and `portal_javascripts`

Through the code changes

To expand Plone using Python, you have to create your own add-on product. Add-on products are distributed as packaged Python modules called *eggs*.

The recommended way is to use the *bobtemplates.plone* command to generate an add-on product skeleton which you can use as a starting point for your development.

Schema-driven forms

This tutorial covers how to build schema-driven forms, using the `z3c.form` and `plone.autoform` libraries.

Introduction

What is `z3c.form` all about?

HTML forms are the cornerstone of modern web applications. When you interact with Plone, you use forms all the time - to search the content store, to edit content items, to fill in your personal details. You will notice that most of these forms use the same layout and conventions, and that they all rely on common patterns such as server-side validation and different buttons resulting in different actions.

Over the years, several approaches have evolved to deal with forms. A few of the most important ones are:

- Creating a simple view with an HTML form that submits to itself (or another view), where the request is validated and processed in custom Python code. This is very flexible and requires little learning, but can also be fairly cumbersome, and it is harder to maintain a common look and feel and behaviour across all forms. See the [Views and viewlets](#) for some hints on one way to build such views.
- Using the `CMFFormController` library. This relies on special page objects known as “controller page templates” that submit to “controller python scripts”. The form controller takes care of the flow between forms and actions, and can invoke validator scripts. This only superficially addresses the creation of standard form layouts and widgets, however. It is largely deprecated, although Plone still uses it internally in places.
- Using `zope.formlib`. This is a library which ships with Zope. It is based on the principle that a *schema interface* defines a number of form fields, constraints and so on. Special views are then used to render these using a standard set of widgets. Formlib takes care of page flow, validation and the invocation of *actions* - methods that correspond to buttons on the form. Formlib is used for Plone’s control panels and portlets. However, it can be cumbersome to use, especially when it comes to creating custom widgets or more dynamic forms.
- Using `‘z3c.form’_`. This is a newer library, inspired by formlib, but more flexible and modern.

This manual will show you how to use `z3c.form` in a Plone context. It will use tools and patterns that are consistent with those used for Dexterity development, as shown in the Dexterity developer manual, but the information contained herein is not Dexterity specific. Note that Dexterity’s standard add and edit forms are all based on `z3c.form`.

Tools

As a library, `z3c.form` has spawned a number of add-on modules, ranging from new field types and widgets, to extensions that add functionality to the forms built using the framework. We will refer to a number of packages in this tutorial. The most important packages are:

- `z3c.form` itself, the basic form library. This defines the standard form view base classes, as well the default widgets. The `z3c.form` [documentation](#) applies to the forms created here, but some of the packages below simplify or enhance the integration experience.
- `plone.z3cform` makes `z3c.form` usable in Zope 2. It also adds a number of features useful in Zope 2 applications, notably a mechanism to extend or modify the fields in forms on the fly.
- `plone.app.z3cform` configures `z3c.form` to use Plone-looking templates by default, and adds few services, such as a widget to use Plone’s visual editor and “inline” on-the-fly validation of forms. This package must be installed for `z3c.form`-based forms to work in Plone.
- `plone.autoform` improves `z3c.form`’s ability to create a form from a schema interface. By using the base classes in this package, schemata can be more self-describing, for example specifying a custom widget, or specifying relative field ordering. We will use `plone.autoform` in this tutorial to simplify form setup.

- `plone.directives.form` provides tools for registering forms using convention-over-configuration instead of ZCML. We will use *plone.directives.form* to configure our forms in this manual.

A note about versions

This manual is targeted at Plone 4.1 and above (Zope 2.13).

Creating a simple form

Creating A Package

Giving our forms a home

For the purposes of this tutorial, we will create a simple package that adds the necessary dependencies.

If you have an existing package that requires a form, you should be able to add the same dependencies.

For details about creating new packages, see *Bootstrapping Plone add-on development*.

Note: Using `paster` is deprecated instead you should use *bobtemplates.plone*

Deprecated since version may_2015: Use *bobtemplates.plone* instead

First, we create a new package in src:

```
../bin/mrbob -O example.form bobtemplates:plone_addon
```

We create a package from the *Basic* template for Plone 5.0-latest.

We will add `example.form` later as development egg to our buildout. Before we use the autogenerated buildout of the package itself.

Take a look at `buildout.cfg` at the top level of our newly created package. You will find there various useful things:

- instance with your package added to the eggs
- code analysis
- a test runner
- even a robot test runner
- and a releaser

That is everything you need for development. Let us use this buildout.

```
cd example.form/  
python bootstrap-buildout.py  
bin/buildout
```

Let us test it!

```
bin/test -s example.form  
bin/test -s example.form -t test_example.robot --all
```

Our package shall add a form to our Plone site. We use `plone.app.z3cform` to develop the form.

That is why we add it to `install_requires` in `setup.py`


```

# -*- coding: utf-8 -*-
"""Installer for the example.form package."""

from setuptools import find_packages
from setuptools import setup

long_description = (
    open('README.rst').read()
    + '\n' +
    'Contributors\n'
    '=====\n'
    + '\n' +
    open('CONTRIBUTORS.rst').read()
    + '\n' +
    open('CHANGES.rst').read()
    + '\n')

setup(
    name='example.form',
    version='0.1',
    description="An add-on for Plone",
    long_description=long_description,
    # Get more from http://pypi.python.org/pypi?%3Aaction=list_classifiers
    classifiers=[
        "Environment :: Web Environment",
        "Framework :: Plone",
        "Framework :: Plone :: 5.0-latest",
        "Programming Language :: Python",
        "Programming Language :: Python :: 2.7",
    ],
    keywords='Python Plone',
    author='John Doe',
    author_email='john@doe.org',
    url='http://pypi.python.org/pypi/example.form',
    license='GPL',
    packages=find_packages('src', exclude=['ez_setup']),
    namespace_packages=['example'],
    package_dir={'': 'src'},
    include_package_data=True,
    zip_safe=False,
    install_requires=[
        'plone.api',
        'setuptools',
        'z3c.jbot',
        'plone.app.z3cform',
    ],
    extras_require={
        'test': [
            'plone.app.testing',
            'plone.app.contenttypes',
            'plone.app.robotframework[debug]',
        ],
    },
    entry_points="""
    [z3c.autoinclude.plugin]
    target = plone

```

```
    """  
)
```

and add `plone.app.z3cform`'s import step to our profile's `metadata.xml` for an automated installation.

```
<metadata>  
  <version>1000</version>  
  <dependencies>  
    <dependency>profile-plone.app.z3cform:default</dependency>  
  </dependencies>  
</metadata>
```

We have omitted large parts of the buildout configuration here.

The important things to note are:

- We have created a plone 5 add-on using `mr.bob`.
- We have tested the egg in a Plone test environment using the autogenerated `buildout.cfg` of our package.
- We have added dependencies to the egg.

Creating a schema

The starting point for our form

With the form package created and installed, we can create our form schema. Later in this manual, we will cover in more detail how you can perform to configure custom widgets, set up hidden fields and onimperatively in Python code.

The example we'll use for this form is a pizza ordering form. We'll build on this form over the coming sections, so if you look at the example source code, you may find a few extra bits. However, the basics are simple enough.

We'll create a module called `order.py` inside our package (*example/dexterityforms/order.py*), and add the following code to it:

```
from plone.autoform.form import AutoExtensibleForm  
from zope import interface  
from zope import schema  
from zope import component  
from z3c.form import form, button  
  
from Products.statusmessages.interfaces import IStatusMessage  
from example.form import _  
  
class OrderFormSchema(interface.Interface):  
    name = schema.TextLine(  
        title=_(u"Your full name"),  
    )  
  
    address1 = schema.TextLine(  
        title=_(u"Address line 1"),  
    )  
  
    address2 = schema.TextLine(  
        title=_(u"Address line 2"),  
        required=False,
```

```

    )

    postcode = schema.TextLine(
        title=_("Postcode"),
    )

    telephone = schema.ASCIILine(
        title=_("Telephone number"),
        description=_("We prefer a mobile number"),
    )

    orderItems = schema.Set(
        title=_("Your order"),
        value_type=schema.Choice(values=[_("Margherita"),
                                         _("Pepperoni"),
                                         _("Hawaiian")])
    )

```

For now, this form is quite simple. The list of pizzas is hard-coded, and we can only choose one of each type. We will make it (a little) more realistic later by adding a better vocabulary, creating a custom widget for the pizza order part, and improving the look and feel with a custom template.

At the top, we have included a number of imports. Some of these pertain to the form view, which will be described next. Other than that, we have simply defined a schema that describes the form's fields. The *title* and *description* of each field are used as label and help text, respectively. The *required* attribute can be set to *False* for optional fields. For a full field and widgets reference, see the Dexterity developer manual. (It is no accident that the Dexterity content type fields and widgets are defined in the same manner as those of a standalone form!)

Also notice how all the user-facing strings are wrapped in the message factory to make them translatable. The message factory is imported as `_`, which helps tools like *gettext* extract strings for translation. If you are sure your form will never need to be translated, you can skip the message factory in *interfaces.py* and use plain unicode strings, i.e. `u"Postcode"` instead of `_(u"Postcode")`

Create a generic adapter to fill the form from anywhere

```

class OrderFormAdapter(object):
    interface.implements(OrderFormSchema)
    component.adapts(interface.Interface)

    def __init__(self, context):
        self.name = None
        self.address1 = None
        self.address2 = None
        self.postcode = None
        self.telephone = None
        self.orderItems = None

```

We are almost done with our most basic form. Before we can use the form, however, we need to create a form view and define some actions (buttons). That is the subject of the next section.

Creating the form view

Using our schema in a form

To render our form, we need to create a view that uses a *z3c.form* base class. The view is registered like any other in ZCML. It is then configured with the schema to use for form fields, the label (page title) and description (lead-in text) to show, and actions to render as buttons.

Still in *order.py*, we add the following:

```
class OrderForm(AutoExtensibleForm, form.Form):
    schema = OrderFormSchema
    form_name = 'order_form'

    label = _(u"Order your pizza")
    description = _(u"We will contact you to confirm your order and delivery.")

    def update(self):
        # disable Plone's editable border
        self.request.set('disable_border', True)

        # call the base class version - this is very important!
        super(OrderForm, self).update()

    @button.buttonAndHandler(_(u'Order'))
    def handleApply(self, action):
        data, errors = self.extractData()
        if errors:
            self.status = self.formErrorsMessage
            return

        # Handle order here. For now, just print it to the console. A more
        # realistic action would be to send the order to another system, send
        # an email, or similar

        print u"Order received:"
        print u"  Customer: ", data['name']
        print u"  Telephone:", data['telephone']
        print u"  Address:   ", data['address1']
        print u"                ", data['address2']
        print u"                ", data['postcode']
        print u"  Order:    ", ', '.join(data['orderItems'])
        print u""

        # Redirect back to the front page with a status message

        IStatusMessage(self.request).addStatusMessage(
            _(u"Thank you for your order. We will contact you shortly"),
            "info"
        )

        contextURL = self.context.absolute_url()
        self.request.response.redirect(contextURL)

    @button.buttonAndHandler(_(u"Cancel"))
    def handleCancel(self, action):
        """User cancelled. Redirect back to the front page.
        """
        contextURL = self.context.absolute_url()
        self.request.response.redirect(contextURL)
```

The form is registered in *configure.zcml*

```
<configure
    xmlns="http://namespaces.zope.org/zope"
    xmlns:browser="http://namespaces.zope.org/browser"
    xmlns:plone="http://namespaces.plone.org/plone"
```

```

i18n_domain="example.form">

<!-- Set overrides folder for Just-a-Bunch-Of-Templates product -->
<include package="z3c.jbot" file="meta.zcml" />
<browser:jbot
    directory="overrides"
    layer="example.form.interfaces.IExampleFormLayer"
/>

<!-- Publish static files -->
<browser:resourceDirectory
    name="example.form"
    directory="static"
/>

<adapter factory=".order.OrderFormAdapter"/>

<browser:page
    for="Products.CMFCore.interfaces.ISiteRoot"
    name="order-pizza"
    class=".order.OrderForm"
    permission="zope2.View"
/>

</configure>

```

Let's go through this in some detail:

- We derive our form view from one of the standard base classes in *plone.autoform*. It comes without any of the standard actions that can be found on more specialised base classes such as *SchemaAddForm* or *SchemaEditForm*. It basically mirrors the *z3c.form.form.Form* base class.
- Next, we specify the schema via the *schema* attribute.
- We set *ignoreContext* to *True*. This tells *z3c.form* not to attempt to read the current value of any of the form fields from the context. The default behaviour is to attempt to adapt the context (the Plone site root in this case) to the schema interface and read the schema attribute value from this adapter when first populating the form. This makes sense for edit forms and things like control panels, but not for a standalone form like this.
- We then set a *label* and *description* for the form. In the standard form template, these are rendered as a page header and lead-in text, respectively.
- We override the *update()* method to set the *disable_border* request variable. This hides the editable border when rendering the form. We then call the base class version of *update()*. This is crucial for the form to work! *update()* is a good place to perform any pre-work before the form machinery kicks in (before calling the base class version) or post-processing afterwards (after calling the base class version). See the section on the form rendering lifecycle later in this manual for the gory details.
- Finally, we define two actions, using the *@button.buttonAndHandler()* decorator. Each action is rendered as a button (in order). The argument is a (translated) string that will be used as a button label. The decorated handler function will be called when the button is clicked.
- We then use the standard way to register the view via *zcml*: *name* gives it a friendly name (used as a path segment in the URL); *for* sets the type of context where the form is available (here, we make it available on the Plone site root, though any interface or class may be passed; to make the form available on any context, use *** as *for*); *permission* specifies a permission which the user must have to be able to view the form (here, we use the standard *zope2.View* permission).

For the purposes of this test, the actual work we do with the main handler is relatively contrived. However, the patterns

are generally applicable.

The second button (cancel) is the simpler of the two. It performs no validation and simply redirects to the context's default view, i.e. the portal front page in this case.

The first button actually extracts the data from the form, using `self.extractData()`. This returns a tuple of the form data, which has been converted to the field's underlying type by each widget (thus, the value corresponding to the *Set* field contains a *set*) and any errors. If there are errors, we abort, setting `self.status` to confer an error message at the top of the page. Otherwise, we use the form data (here just printing the output to the console - you need to run Zope in foreground mode to see these messages), add a cookie-tracked status message (so that it can appear on the next page) and redirect the user to the context's default view. In this case, that means the portal front page.

Testing the form

Seeing the form in action

The schema and (grokked) form view is all that's needed to create the first iteration of the form. We can now install our new package and test the form.

First, we make sure that we have run `bin/buildout` so that the new package is available to the Zope instance script. We then start up Zope in foreground mode:

```
bin/instance fg
```

Next, we create or go to a Plone site, and install the new *Example forms* product via the new Plone site creation form or the Add-ons control panel. This should also install the product called *Plone z3c.form support* (from the *plone.app.z3cform* package) as a dependency.

We haven't created any links to the form yet (though you could do so in a content item or portlet by inserting a manually-entered URL), but the form can be visited by going to the `@@order-pizza` view on the Plone site root, e.g.:

`http://localhost:8080/Plone/@@order-pizza`

It should look something like this:

Try to fill in the form and use the two buttons. You should see the validation (both on-the-fly and after submit if you ignore the on-the-fly warnings), as well a message printed to the console if a valid form is submitted when clicking the *Order* button.

Remember: We have worked so far in a development environment of the package itself. Now you want to insert `example.form` to your project's buildout. It should look just similar to the packages buildout.

```
[buildout]
extends = http://dist.plone.org/release/5.0-latest/versions.cfg
extensions = mr.developer
parts =
    instance
    test
    code-analysis
    releaser
develop = src/example.form

[instance]
recipe = plone.recipe.zope2instance
user = admin:admin
http-address = 8080
eggs =
    Plone
```

Order your pizza

We will contact you to confirm your order and delivery.

Your full name ●

Address line 1 ●

Address line 2

Postcode ●

Telephone number ● *We prefer a mobile number*

Your order ●

Margherita
Pepperoni
Hawaiian

```
Pillow
example.form [test]
```

[code-analysis]

```
recipe = plone.recipe.codeanalysis
directory = ${buildout:directory}/src/example
flake8-exclude = bootstrap.py,bootstrap-buildout.py,docs,*.egg.,omelette
flake8-max-complexity = 15
flake8-extensions =
    flake8-blind-except
    flake8-debugger
    flake8-coding
```

[omelette]

```
recipe = collective.recipe.omelette
eggs = ${instance:eggs}
```

[test]

```
recipe = zc.recipe.testrunner
eggs = ${instance:eggs}
defaults = ['-s', 'example.form', '--auto-color', '--auto-progress']
```

[robot]

```
recipe = zc.recipe.egg
eggs =
    ${test:eggs}
    plone.app.robotframework[debug,ride,reload]
```

[releaser]

```
recipe = zc.recipe.egg
eggs = zest.releaser
```

[versions]

```
setuptools = 18.0.1
zc.buildout = 2.2.5
zc.recipe.egg = 2.0.1

flake8 = 2.3.0

robotframework = 2.8.4
robotframework-ride = 1.3
robotframework-selenium2library = 1.6.0
robotsuite = 1.6.1
selenium = 2.46.0
```

[sources]

```
example.form = fs example.form
```

Customising form behaviour

Validation

How to validate a form prior to processing

All forms apply some form of validation. In *z3c.form*, validators can be executed in action handlers. If the validation fails, the action handler can choose how to proceed. For “submit” type buttons, that typically means showing error messages next to the relevant form fields. For “cancel” type buttons, the validation is normally skipped entirely.

Form validation takes two forms: field-level validation, pertaining to the value of an individual field, and form-level validation, pertaining to the form as a whole. Form-level validation is less common, but can be useful if fields have complex inter-dependencies.

Field-level validation

The simplest field-level validation is managed by the fields themselves. All fields support a *required* attribute, defaulting to *True*. The default field validator will return an error if a required field is not filled in. Some fields also support more specific properties that affect validation:

- Text fields like *Bytes*, *BytesLine*, *ASCII*, *ASCIILine*, *Text* and **TextLine*, as well as sequence fields like *Tuple*, *List*, *Set*, **FrozenSet* and *Dict* all support two properties, *min_length* and *max_length*, which can be set to control the minimum and maximum allowable length of the field’s value.
- Numeric fields like *Int*, *Float* and *Decimal*, as well as temporal fields like *Date*, *Datetime* and *Timedelta* all support two properties, *min* and *max*, setting minimum and maximum (inclusive) allowable values. In this case, the min/max value needs to be of the same type as field, so for an *Int* field, the value of this property is an integer, whereas for a *Datetime* field, it is a Python *datetime* object.
- A *Choice* field only allows values in a particular vocabulary. We will cover vocabularies in the next section.

Constraints

If you require more specific validation, and you have control over the schema, you can specify a *constraint* function. This will be passed the submitted value (which is converted to a value appropriate for the field, so that e.g. a *List* field is passed a list). If the value is acceptable, the function should return *True*. If not, it should raise a *zope.schema.Invalid* exception or a derivative (returning *False* will also result in an error, but one without a meaningful error message).

Here is the order form schema again, this time with a constraint function:

```
from plone.supermodel import model

from zope.interface import Invalid
from zope import schema
from z3c.form import button

from Products.CMFCore.interfaces import ISiteRoot
from Products.statusmessages.interfaces import IStatusMessage

from example.dexterityforms.interfaces import MessageFactory as _

def postcodeConstraint(value):
    """Check that the postcode starts with a 6
    """
    if not value.startswith('6'):
        raise Invalid(_(u"We can only deliver to postcodes starting with 6"))
    return True
```

```
class IPizzaOrder(model.Schema):

    name = schema.TextLine(
        title=_("Your full name"),
    )

    address1 = schema.TextLine(
        title=_("Address line 1"),
    )

    address2 = schema.TextLine(
        title=_("Address line 2"),
        required=False,
    )

    postcode = schema.TextLine(
        title=_("Postcode"),
        constraint=postcodeConstraint,
    )

    telephone = schema.ASCIILine(
        title=_("Telephone number"),
        description=_("We prefer a mobile number"),
    )

    orderItems = schema.Set(
        title=_("Your order"),
        value_type=schema.Choice(values=[_(u'Margherita'), _(u'Pepperoni'), _(u
↪ 'Hawaiian')]))
    )
```

Notice how the `postcodeConstraint()` function is passed a value (a unicode string in this case, since the field is a `*TextLine`), which we validate. If we consider the value to be invalid, we raise an `Invalid` exception, with the error message passed as the exception argument. Otherwise, we return `True`.

Field widget validators

Constraints are relatively easy to write, but they have two potential drawbacks: First of all, they require that we change the underlying interface. This is no problem if the interface exists only for the form, but could be a problem if it is used in other contexts as well. Second, if we want to re-use a validator for multiple forms, we would need to modify multiple schemata.

z3c.form's field widget validators address these shortcomings. These are specific to the form; by contrast, constraints are a feature of `zope.interface` interfaces and apply in other scenarios where interfaces are used as well.

For example:

```
from plone.directives import form
from example.dexterityforms.interfaces import MessageFactory as _
from plone.supermodel import model
from z3c.form import validator
from zope import schema
import zope.component
import zope.interface

...
```

```

@form.validator.validator(field=IPizzaOrder['phone_number'])
class IPizzaOrder(model.Schema):

    phone_number = schema.TextLine(
        title=_("Phone number"),
        description=_("Your phone number in international format. E.g. +44 12 123_
↪1234"),
        required=False,
        default=u"")

class PhoneNumberValidator(validator.SimpleFieldValidator):
    """ z3c.form validator class for international phone numbers """

    def validate(self, value):
        """ Validate international phone number on input """
        super(PhoneNumberValidator, self).validate(value)

        allowed_characters = "+- () / 0123456789"

        if value != None:

            value = value.strip()

            if value == "":
                # Assume empty string = no input
                return

            # The value is not required
            for c in value:
                if c not in allowed_characters:
                    raise zope.interface.Invalid(_(u"Phone number contains bad_
↪characters"))

            if len(value) < 7:
                raise zope.interface.Invalid(_(u"Phone number is too short"))

# Set conditions for which fields the validator class applies
validator.WidgetValidatorDiscriminators(PhoneNumberValidator, field=IPizzaOrder[
↪'phone_number'])

# Register the validator so it will be looked up by z3c.form machinery
zope.component.provideAdapter(PhoneNumberValidator)

```

This registers an adapter, extending the SimpleFieldValidator base class, and calling the superclass version of validate() to gain the default validation logic. In the validate() method, we can use variables like self.context, self.request, self.view, self.field and self.widget to access the adapted objects. The WidgetValidatorDiscriminators class takes care of preparing the adapter discriminators.

The valid values for WidgetValidatorDiscriminators are:

context The form's context, typically an interface. This allows a validator to be invoked only on a particular type of content object.

request The form's request. Normally, this is used to specify a browser layer.

view The form view itself. This allows a validator to be invoked for a particular type of form. As with the other

options, we can pass either a class or an interface.

field A field instance, as illustrated above, or a field *type*, e.g. an interface like *zope.schema.IInt*.

widget The widget being used for the field

It is important to realise that if we don't specify the *field* discriminator, or if we pass a field type instead of an instance, the validator will be used for all fields in the form (of the given type).

Form-level validation

Form level validation is less common than field-level validation, but is useful if your fields are inter-dependent in any ways. As with field-level validation, there are two options:

- Invariants are specified at the interface level. As such, they are analogous to constraints.
- Widget manager validators are standalone adapters that are specific to *z3c.form*. As such, they are analogous to field widget validators.

Invariants

Invariants work much like constraints, in that they are called during the form validation cycle and may raise *Invalid* exceptions to indicate a validation problem. Because they are not tied to fields specifically, an error resulting from an invariant check is displayed at the top of the form.

Invariants are written as functions inside the interface definition, decorated with the *zope.interface.invariant* decorator. They are passed a data object that provides the schema interface. In the case of a *z3c.form* form, this is actually a special object that provides the values submitted in the request being validated, rather than an actual persistent object.

For example:

```
from example.dexterityforms.interfaces import MessageFactory as _
from plone.supermodel import model
from zope import schema
from zope.interface import invariant, Invalid

...

class IPizzaOrder(model.Schema):

    name = schema.TextLine(
        title=_("Your full name"),
    )

    address1 = schema.TextLine(
        title=_("Address line 1"),
    )

    address2 = schema.TextLine(
        title=_("Address line 2"),
        required=False,
    )

    postcode = schema.TextLine(
        title=_("Postcode"),
        constraint=postcodeConstraint,
    )
```

```

telephone = schema.ASCIILine(
    title=_("Telephone number"),
    description=_("We prefer a mobile number"),
)

orderItems = schema.Set(
    title=_("Your order"),
    value_type=schema.Choice(values=[_(u'Margherita'), _(u'Pepperoni'), _(u
↪ 'Hawaiian')]))

@invariant
def addressInvariant(data):
    if data.address1 == data.address2:
        raise Invalid(_("Address line 1 and 2 should not be the same!"))

```

Here we have defined a single invariant, although there is no limit to the number of invariants that you can use.

Widget manager validators

Invariants have most of the same benefits and draw-backs as constraints: they are easy to write, but require modifications to the schema interface, and cannot be generalised beyond the interface. Not surprisingly therefore, *z3c.form* provides another option, in the form of a widget manager validator. This is a multi-adapter for (*context*, *request*, *view*, *schema*, *widget manager*) providing *z3c.form.interfaces.IManagerValidator*. The default simply checks invariants, although you can register your own override.

That said, overriding the widget manager validator is not particularly common, because if you need full-form validation and you don't want to use invariants, it is normally easier to place validation in the action handler, as we will see next.

Invoking validators

Unlike some of the earlier form libraries, *z3c.form* does not automatically invoke validators on every form submit. This is actually a good thing, because it makes it much easier to decide when validation makes sense (e.g. there is no need to validate a “cancel” button).

We have already seen the most common pattern for invoking validation in our handler for the “order” button:

```

@button.buttonAndHandler(_(u'Order'))
def handleApply(self, action):
    data, errors = self.extractData()
    if errors:
        self.status = self.formErrorsMessage
        return

    # Handle order here. For now, just print it to the console. A more
    # realistic action would be to send the order to another system, send
    # an email, or similar

    ...

```

Notice how we call *extractData()*, which returns both a dictionary of the submitted data (for valid fields, converted to the underlying field value type) and a dictionary of errors (which is empty if all fields are valid).

Validating in action handlers

Sometimes, it may be useful to perform additional validation in the action handler itself. We can inspect the *data* dictionary, as well as any other aspect of the environment (like *self.context*, the context content object, or *self.request*, the request), to perform validation.

To signal an error, we use one of two exception types:

- `z3c.form.interfaces.ActionExecutionError`, for generic, form-wide errors
- `z3c.form.interfaces.WidgetActionExecutionError`, for field/widget-specific errors

In both cases, these exceptions wrap an *Invalid* exception. Let’s add two examples to our action handler.

```
from example.dexterityforms.interfaces import MessageFactory as _
from plone.supermodel import model
from Products.CMFCore.interfaces import ISiteRoot
from Products.statusmessages.interfaces import IStatusMessage
from z3c.form.interfaces import ActionExecutionError, WidgetActionExecutionError
from zope import schema
from zope.interface import invariant, Invalid
import plone.autoform
import z3c.form

...

class OrderForm(plone.autoform.form.AutoExtensibleForm, z3c.form.form.Form):

    ...

    @z3c.form.button.buttonAndHandler(_(u'Order'))
    def handleApply(self, action):
        data, errors = self.extractData()

        # Some additional validation
        if 'address1' in data and 'address2' in data:

            if len(data['address1']) < 2 and len(data['address2']) < 2:
                raise ActionExecutionError(Invalid(_(u"Please provide a valid address
↪")))

            elif len(data['address1']) < 2 and len(data['address2']) > 10:
                raise WidgetActionExecutionError(
                    'address2',
                    Invalid(u"Please put the main part of the address in the first_
↪field"))

        if errors:
            self.status = self.formErrorsMessage
            return
```

Notice how we perform the check after the *extractData()* call, but before the possible premature return in case of validation errors. This is to ensure all relevant errors are displayed to the user. Also note that whilst the invariant is passed an object providing the schema interface, the *data* dictionary is just that - a dictionary. Hence, we use “dot notation” (*data.address1*) to access the value of a field in the invariant, but “index notation” (*data['address1']*)*to access the value of a field in the handler.

Vocabularies

Static and dynamic lists of valid values

The term “vocabulary” here refers to a list of values that are allowable by a given field. In most cases, that implies a field using a selection widget, like a multi-select list box or a drop-down.

Selection fields use the *Choice* field type. To allow the user to select a single value, use a *Choice* field directly:

```
class ISimplePizza(model.Schema):
    topping = schema.Choice(
        title=_("Choose your topping"),
        values=[_(u'Chicken'), _(u'Pepperoni'), _(u'Tomato')]
    )
```

For a multi-select field, use a *List*, *Tuple*, *Set* or *Frozenset* with a *Choice* as the *value_type*:

```
class IPizzaOrder(model.Schema):

    ...

    orderItems = schema.Set(
        title=_("Your order"),
        value_type=schema.Choice(values=[_(u'Margherita'), _(u'Pepperoni'), _(u
→ 'Hawaiian')])
    )
```

The *Choice* field must be passed one of the following arguments, specifying its vocabulary:

- *values* can be used to give a list of static values
- *source* can be used to refer to an *IContextSourceBinder* or *ISource* instance
- *vocabulary* can be used to refer to an *IVocabulary* instance or (more commonly) a string giving the name of an *IVocabularyFactory* named utility.

We’ll now explore various ways in which we can improve on the *orderItems* list.

Static vocabularies

Up until now, we have been using a static vocabulary, passing the list of allowable values as the *values* parameters to the *Choice* field. This is simple, but has a few draw-backs:

- If the vocabulary changes, we have to change the interface code.
- There is no way to separate the label that the user sees in the selection widget from the value that is extracted.

Dynamic vocabularies

To implement a more dynamic vocabulary, we can use a source. Before we do that, though, let’s consider where our data will come from.

We want to make the “order items” list more dynamic, and allow the list of available pizza types to be managed through the web. There are various ways to do this, including modelling pizzas as content items and creating a source that performs a catalog query to find them all. To manage a simple list, however, we can use *plone.app.registry* and install the list with our product’s extension profile. An administrator could then use the registry control panel to change the list. We won’t go into *plone.app.registry* in detail here, but you can read its [documentation](#) to get a full understanding of what it is and how it works.

First, we need to add *plone.app.registry* as a dependency in *setup.py*:

```
install_requires=[
    'setuptools',
    'plone.app.z3cform',
    'plone.directives.form',
    'plone.app.registry',
],
```

We also want to configure it when our product is installed in Plone, so we edit *profiles/default/metadata.xml* as follows:

```
<metadata>
  <version>1</version>
  <dependencies>
    <dependency>profile-plone.app.z3cform:default</dependency>
    <dependency>profile-plone.app.registry:default</dependency>
  </dependencies>
</metadata>
```

Next, we create a *registry.xml* containing the following:

```
<registry>

  <record name="example.dexterityforms.pizzaTypes">
    <field type="plone.registry.field.Tuple">
      <title>Pizza types</title>
      <value_type type="plone.registry.field.TextLine" />
    </field>
    <value>
      <element>Margherita</element>
      <element>Pepperoni</element>
      <element>Hawaiian</element>
    </value>
  </record>

</registry>
```

After re-running buildout and (re-)installing our product in the

Terminology

When working with dynamic vocabularies, we come across some terminology that is worth explaining:

- A *term* is an entry in the vocabulary. The term has a value. Most terms are *tokenised* terms which also have a token, and some terms are *titled*, meaning they have a title that is different to the token.
- The *token* must be an ASCII string. It is the value passed with the request when the form is submitted. A token must uniquely identify a term.
- The *value* is the actual value stored on the object. This is not passed to the browser or used in the form. The value is often a unicode string, but can be any type of object.
- The *title* is a unicode string or translatable message. It is used in the form and displayed to the user.

One-off sources with a context source binder

We can make a one-off dynamic vocabulary using a context source binder. This is simply a callable (usually a function or an object with a `__call__` method) that provides the `IContextSourceBinder` interface and takes a *context* parameter. The *context* argument is the context of the form view. The callable should return a vocabulary, which is achieved by using the `SimpleVocabulary` class from `zope.schema`.

Here is an example that returns our pizza types:

```
from five import grok
from plone.supermodel import model
from plone.directives import form

from zope.component import queryUtility

from zope import schema

from zope.schema.interfaces import IContextSourceBinder
from zope.schema.vocabulary import SimpleVocabulary

from plone.registry.interfaces import IRegistry

...

@grok.provider(IContextSourceBinder)
def availablePizzas(context):
    registry = queryUtility(IRegistry)

    terms = []

    if registry is not None:
        for pizza in registry.get('example.dexterityforms.pizzaTypes', ()):
            # create a term - the arguments are the value, the token, and
            # the title (optional)
            terms.append(SimpleVocabulary.createTerm(pizza, pizza.encode('utf-8'),
            ↪pizza))

    return SimpleVocabulary(terms)
```

Here, we have defined a function acting as the `IContextSourceBinder`, as specified via the `@grok.provider()` decorator. This looks up the registry and looks for the record we created with *registry.xml* above (remember to re-install the product in the Add-on control panel or the *portal_quickinstaller* tool if you modify this file). We then use the `SimpleVocabulary` helper class to create the actual vocabulary.

The `SimpleVocabulary` class additionally contains two class methods that can be used to create vocabularies from lists:

- `fromValues()` takes a simple list of values and returns a tokenised vocabulary where the values are the items in the list, and the tokens are created by calling `str()` on the values.
- `fromItems()` takes a list of (*token*, *value*) tuples and creates a tokenised vocabulary with the token and value specified.

We can also instantiate a `SimpleVocabulary` directly and pass a list of terms in the initialiser as we have done above. The `createTerm()` class method can be used to create a term from a *value*, *token* and *title*. Only the value is required.

To use this context source binder, we use the *source* argument to the *Choice* constructor:

```
class IPizzaOrder(model.Schema):
```

```
...

orderItems = schema.Set(
    title=_("Your order"),
    value_type=schema.Choice(source=availablePizzas)
)
```

Parameterised sources

Sometimes, it is useful to parameterise the source. For example, we could generalise the pizza source to work with any registry value containing a sequence, by passing the registry key as an argument. This would allow us to create many similar vocabularies and call upon them in code.

This degree of generalisation is probably overkill for our use case, but to illustrate the point, we'll outline the solution below.

First, we turn our *IContextSourceBinder* into a class that is initialised with the registry key

```
class RegistrySource(object):
    grok.implements(IContextSourceBinder)

    def __init__(self, key):
        self.key = key

    def __call__(self, context):
        registry = queryUtility(IRegistry)
        terms = []

        if registry is not None:
            for value in registry.get(self.key, ()):
                terms.append(SimpleVocabulary.createTerm(value, value.encode('utf-8'),
→ value))

        return SimpleVocabulary(terms)
```

Notice how in our first implementation, the function *provided* the *IContextSourceBinder* interface, but the class here *implements* it. This is because the function was the context source binder callable itself. Conversely, the class is a factory that creates *IContextSourceBinder* objects, which in turn are callable.

Again, the source is set using the *source* argument to the *Choice* constructor.

```
orderItems = schema.Set(
    title=_("Your order"),
    value_type=schema.Choice(source=RegistrySource('example.dexterityforms.
→pizzaTypes'))
)
```

When the schema is initialised on startup, the a *RegistrySource* object is instantiated, storing the desired registry key in an instance variable. Each time the vocabulary is needed, this object will be called (i.e. the *__call__()* method is invoked) with the context as an argument, and is expected to return an appropriate vocabulary.

Named vocabularies

Context source binders are great for simple dynamic vocabularies. They are also re-usable, since we can import the source from a single location and use it in multiple instances. However, we may want to provide an additional level of

decoupling, by locating a vocabulary by name, not necessarily caring where or how it is implemented.

Named vocabularies are similar to context source binders, but are components registered as named utilities, referenced in the schema by name only. This allows local overrides of the vocabulary via the Component Architecture, and makes it easier to distribute vocabularies in third party packages.

Note: Named vocabularies cannot be parameterised in the way as we did with the context source binder above, since they are looked up by name only.

We can turn our first dynamic vocabulary into a named vocabulary by creating a named utility providing *IVocabularyFactory*, like so:

```
from five import grok
from zope.component import queryUtility

from zope import schema
from zope.schema.interfaces import IVocabularyFactory

from zope.schema.vocabulary import SimpleVocabulary

from plone.registry.interfaces import IRegistry


class PizzasVocabulary(object):
    grok.implements(IVocabularyFactory)

    def __call__(self, context):
        registry = queryUtility(IRegistry)
        terms = []
        if registry is not None:
            for pizza in registry.get('example.dexterityforms.pizzaTypes', ()):
                # create a term - the arguments are the value, the token, and
                # the title (optional)
                terms.append(SimpleVocabulary.createTerm(pizza, pizza.encode('utf-8'),
→ pizza))
            return SimpleVocabulary(terms)
grok.global_utility(PizzasVocabulary, name=u"example.dexterityforms.availablePizzas")
```

Note: By convention, the vocabulary name is prefixed with the package name, to ensure uniqueness.

We can make use of this vocabulary in any schema by passing its name to the *vocabulary* argument of the *Choice* field constructor:

```
orderItems = schema.Set(
    title=(u"Your order"),
    value_type=schema.Choice(vocabulary='example.dexterityforms.availablePizzas')
)
```

As you might expect, there are a number of standard vocabularies that come with Plone and third party packages, most of which are named vocabularies. Many of these can be found in the *plone.app.vocabularies* package, and add-ons such as *plone.principalsource*.

Widgets

Changing the widget used to render a field

Like most form libraries, *z3c.form* separates a field – a representation of the value being provided by the form – from its widget – a UI component that renders the field in the form. In most cases, the widget is rendered as a simple HTML `<input />` element, although more complex widgets may use more complex markup.

The simplest widgets in *z3c.form* are field-agnostic. However, we nearly always work with *field widgets*, which make use of field attributes (e.g. constraints or default values) and optionally the current value of the field (in edit forms) during form rendering.

Most of the time, we don't worry too much about widgets: each of the standard fields has a default field widget, which is normally sufficient. If we need to, however, we can override the widget for a given field with a new one.

Selecting a custom widget using form directives

plone.directives.form provides a convenient way to specify a custom widget for a field, using the *form.widget()* directive:

```
from five import grok
from plone.supermodel import model
from plone.directives import form

from zope import schema

from plone.app.z3cform.wysiwyg import WysiwygFieldWidget

...

class IPizzaOrder(model.Schema):

    ...

    form.widget('notes', WysiwygFieldWidget)
    notes = schema.Text(
        title=(u"Notes"),
        description=(u>Please include any additional notes for delivery"),
        required=False
    )
```

The argument can be either a field widget factory, as shown here, or the full dotted name to one (*plone.app.z3cform.wysiwyg.WysiwygFieldWidget* in this case).

Updating widget settings

All widgets expose properties that control how they are rendered. You can set these properties by passing them to the widget directive:

```
class IPizzaOrder(model.Schema):

    ...

    form.widget('postcode', size=4)
    postcode = schema.TextLine()
```

```
title=_ (u"Postcode"),
)
```

Note: Support for specifying widget properties was added in `plone.autoform` 1.4.

Some of the more useful properties are shown below. These generally apply to the widget’s `<input />` element.

- *class*, a string, can be set to a CSS class.
- *style*, a string, can be set to a CSS style string
- *title*, a string, can be used to set the HTML attribute with the same name
- *onclick*, *ondblclick*, etc can be used to associate JavaScript code with the corresponding events
- *disabled* can be set to `True` to disable input into the field

Other widgets also have attributes that correspond to the HTML elements they render. For example, the default widget for a *Text* field renders a `<textarea />`, and has attributes like *rows* and *cols*. For a *TextLine*, the widget renders an `<input type="text" />`, which supports a *size* attribute, among others.

Take a look at `z3c.form`’s `browser/interfaces.py` for a full list of attributes that are used.

Supplying a widget factory

Later in this manual, we will learn how to set up the *fields* attribute of a form manually, as is done in “plain” `z3c.form`, instead of using the *schema* shortcut that is provided by `plone.autoform`. If you are using this style of configuration (or simply looking at the basic `z3c.form` documentation), the syntax for setting a widget factory is:

```
class OrderForm(Form):

    fields = field.Fields(IPizzaOrder)
    fields['notes'].widgetFactory = WysiwygFieldWidget

    ...
```

Widget reference

You can find the default widgets in the *browser* package in `z3c.form`. The `z3c.form` documentation contains a listing of all the default widgets that shows the HTML output of each.

In addition, the Dexterity manual contains an overview of common custom widgets.

Actions (buttons)

Defining form buttons and executing code when they are clicked

`z3c.form` defines a rich framework for defining, processing and executing *actions*, an abstraction of the “outcome” of a form. Actions are not necessarily related to form buttons, but for the vast majority of use cases, we can think of forms buttons as a special type of widget that represents an underlying action. Such “button actions” are usually the only type of action we will ever use. Actions are nearly always associated with a handler method, which will be called by the framework when a form was submitted in response to a click of a particular button.

The usual way to define actions and buttons is to use the `@button.buttonAndHandler()` decorator. This takes as a minimum the button title as an argument. We have already seen two examples of this in our pizza order form:

```
@button.buttonAndHandler(_(u'Order'))
def handleApply(self, action):
    ...

@button.buttonAndHandler(_(u'Cancel'))
def handleCancel(self, action):
    ...
```

The name of the method is not particularly important, but it needs to be unique. The body of the handler function may react to the button however is appropriate for the form's use case. It may perform a redirect or update form properties prior to re-rendering of the form. It should not return anything. Use the `self.extractData()` helper to return a tuple of the form's submitted data and any errors, as shown in the preceding examples.

The *action* argument is the action that was executed. We normally ignore this, but it may be introspected to find out more about the action. The `isExecuted()` method can be used to determine if the corresponding button was indeed clicked, and would normally be *True* within any action handler that is called by the framework. The *title* attribute contains the button title as shown to the user.

Access keys

To define a HTML access key for a button, use the *accessKey* keyword argument:

```
@button.buttonAndHandler(_(u'Order'), accessKey=u"o")
def handleApply(self, action):
    ...
```

Conditional actions

If a button should only be shown in some cases, we can use the *condition* keyword argument, passing a function that takes as its only parameter the form to which the button belongs. If this does not return *True*, the button will be omitted from the form:

```
...

import datetime

def daytime(form):
    hour = datetime.datetime.now().hour
    return hour >= 9 and hour <= 17:

class MyForm(form.SchemaForm)

    ...

    @button.buttonAndHandler(_(u'Give me a call'), condition=daytime)
    def handleCallbackRequest(self, action):
        ...
```

Updating button properties

As with regular widgets, it is sometimes useful to set properties on buttons after they have been instantiated by `z3c.form`. One common requirement is to add a CSS class to the button. The standard edit form in *plone.dexterity* does this, for example, to add Plone's standard CSS classes. The usual approach is to override `updateActions()`, which is called during the form update cycle:

```
def updateActions(self):
    super(AddForm, self).updateActions()
    self.actions["save"].addClass("context")
    self.actions["cancel"].addClass("standalone")
```

Notice how we call the base class version first to ensure the actions have been properly set up. Also bear in mind that if a button is conditional, it may not be in `self.actions` at all.

Buttons are really just HTML input widgets, so you can set other properties too, including attributes like `onclick` or `ondblclick` to install client-side JavaScript event handlers.

Fieldsets

Breaking forms into multiple fieldsets

`z3c.form` supports the grouping of form fields into what is known as *groups*. A form class may mix in `z3c.form.group.GroupForm` to gain support for groups, setting the `groups` variable to a list of *Group* subclasses. The *Group* base class behaves much like the *Form* base class, but is used only for grouping fields, and cannot have actions.

In Plone, groups are represented as fieldsets. The standard templates make these look like dynamic tabs, much like those we can find in the edit forms for most Plone content. For this reason, *plone.supermodel* provides a directive called `model.fieldset()`, which can be used to create fieldsets.

Note: The `z3c.form Group` idiom is still supported, and can be mixed with the more declarative `model.fieldset()` approach. However, the latter is usually easier to use.

To illustrate fieldsets, let's give customers the option to leave feedback on our pizza ordering form. To keep our main form short, we will put this in a separate fieldset. Note that there is still only one set of submit buttons, i.e. all fieldsets are submitted at once. This is purely for aesthetic effect.

```
from example.dexterityforms.interfaces import MessageFactory as _
from plone.autoform import directives as form
from plone.supermodel import model
from zope import schema

...

class IPizzaOrder(model.Schema):

    # Main form

    name = schema.TextLine(
        title=_(u"Your full name"),
    )

    address1 = schema.TextLine(
        title=_(u"Address line 1"),
    )
```

```
address2 = schema.TextLine(
    title=_("Address line 2"),
    required=False,
)

postcode = schema.TextLine(
    title=_("Postcode"),
    constraint=postcodeConstraint,
)

telephone = schema.ASCIILine(
    title=_("Telephone number"),
    description=_("We prefer a mobile number"),
)

orderItems = schema.Set(
    title=_("Your order"),
    value_type=schema.Choice(source=availablePizzas)
)

form.widget('notes', WysiwygFieldWidget)
notes = schema.Text(
    title=_("Notes"),
    description=_("Please include any additional notes for delivery"),
    required=False
)

# Feedback fieldset

model.fieldset(
    'feedback',
    label=_("Feedback"),
    fields=['feedbackNote', 'feedbackEmail']
)

feedbackNote = schema.Text(
    title=_("Feedback"),
    description=_("Please provide any feedback below"),
    required=False,
)

feedbackEmail = schema.TextLine(
    title=_("Email address"),
    description=_("If you'd like us to contact you, please give us an email_
↪address below"),
    required=False,
)

...
```

Note: Since this approach uses form schema hints, the schema must derive from *model.Schema* and the form base class must extend *plone.autoform.AutoExtensibleForm*. In our example, we are using *SchemaForm*, a subclass of *AutoExtensibleForm*.

Above, we have declared a single fieldset, and listed the fields within it. Those fields not explicitly associated with a

fieldset end up in the “default” fieldset. We also set a fieldset name and label. The label is optional.

It is possible to use the same fieldset name multiple times in the same form. This is often the case when we use the *additional_schemata* property to set secondary schemata for our form. In this case, the *label* from the first *fieldset* directive encountered will be used.

Form types

Base forms and schema forms

Understanding the two types of forms work with in this manual

z3c.form comes with a few base classes for forms, covering common use cases including page forms, edit forms, add forms and display forms. In this manual, we are actually using some intermediary base classes from *plone.directives.form*, which serve two purposes: they allow the forms to be “grokked”, for example to associate a page template or register the form as a view using directives like *grok.context()* and *grok.name()*; some of them also provide a hook for *schema forms*, which use form hints supplied in directives (like *form.widget()* as we saw in the previous section) that are interpreted by *plone.autoform* to configure the form’s fields. Whilst we can do everything in code using the plain *z3c.form* API, many people may prefer the more declarative style of configuration that comes with *plone.autoform* and *plone.directives.form*, because it involves less code and keeps the field-specific form configuration closer to the field definitions.

Over the next several sections, we will discuss the various form base classes. A brief overview follows.

z3c.form.form.BaseForm This base class is not to be used directly, but is the ancestor of all *z3c.form* forms. It defines attributes like *label* (the form’s title), *mode* (the default mode for the form’s fields, usually ‘input’ in regular forms and ‘display’ in display forms), *ignoreContext*, *ignoreRequest* (see below) and *ignoreReadonly* (which omits readonly fields from the form). It also defines the basic *update()* and *render()* methods that are the basis of the form rendering cycle, which we will explain towards the end of this manual, and the *getContent()* helper method which can be used to tell the form about an alternative context - see below.

plone.directives.form.Form (extending z3c.form.form.BaseForm) A basic full-page form. It supports actions (buttons), and will by default read field values from the request (unless *ignoreRequest* is *True*) or the context (unless *ignoreContext* is *True*).

plone.directives.form.SchemaForm This is identical to *Form*, except that it will construct its fields *plone.autoform* schema hints. The *schema* attribute is required, and must be a schema interface. The *additional_schemata* attribute may be set to a tuple of additional schemata - see below.

plone.directives.form.AddForm (extending z3c.form.form.AddForm) A basic content add form with two actions - save and cancel. This implements default Plone semantics for adding content. Note that if you are using Dexterity, you should use the Dexterity add form instead. See the Dexterity documentation for details.

plone.directives.form.SchemaAddForm The schema form equivalent of *AddForm*.

plone.directives.form.EditForm A basic edit form with two actions - save and cancel. This operates on the context returned by the *getContent()* helper method. By default, that’s the context of the form view (*self.context*), but we can override *getContent()* to operate on something else. In particular, it is possible to operate on a dictionary. See the section on edit forms shortly. Note that if you are using Dexterity, you should use the Dexterity edit form instead. See the Dexterity documentation for details.

plone.directives.form.SchemaEditForm The schema form equivalent of *EditForm*.

plone.directives.dexterity.DisplayForm This is a display form view based on the *WidgetsView* base class from *plone.autoform*. You can use this much like *grok.View*, except that it must be initialised with a *schema*, and optionally a tuple of *additional_schemata*. There are several helper variables set during the *update()* cycle which provide easy access to the form’s widgets in display mode.

Context and request

When a form is first rendered, it will attempt to fill fields based on the following rules:

- If *ignoreRequest* is *False* (as is the default for all forms bar display forms), and a value corresponding to the field is in the request, this will be used. This normally means that the form was submitted, but that some validation failed, sending the user back to the form to correct their mistake.
- If no request value was found and *ignoreContext* is *False* (as is the default for all forms bar add forms), the form will look for an associated interface for each widget. This is normally the schema interface of the field that the widget is rendering. It will then attempt to adapt the context to that interface (if the context provides the interface directly, as is often the case for edit and display forms, the context is used as-is). If no such adapter exists, form setup will fail. If this happens, you can either set *ignoreContext* = *True* (which is normally appropriate for free-standing forms like the examples earlier in this manual), supply an adapter (which is normally appropriate for forms that edit some aspect of the context), or override *getContent()* to return a content that is adaptable to the schema interface.
- If no request or context value was found and the field has a default value, this will be used.

Primary and additional schemata in schema forms

When using a schema form, it is possible to set two form properties supplying schemata for the form:

- *schema* is required for all schema forms, and must point to a schema interface. This is known as the default or primary schema for the form.
- *additional_schemata* is optional, and can be set to a tuple or list of schema interfaces. These will also be included in the form.

Note: If you want to make the schema dynamic, you can implement these as read-only properties. This is how Dexterity’s add and edit forms work, for example - they look up the primary schema from the type information in *portal_types*, and additional schemata from behaviours.

Later in this manual, we will learn about creating tabbed fieldsets, also known as groups. The schema forms support a property *autoGroups* which default to *False*. When set to *True*, the primary schema will be used as the primary fieldset, and each schema in *additional_schemata* will become its own fieldset. The schema name will become the fieldset name, and its docstring will become its description. This is somewhat inflexible, but can be useful for certain forms where the fieldsets need to be dynamically looked up.

Page forms

The most basic type of form

A page form, or simply “form”, is a basic, “standalone” form. The pizza order example in this manual is a page form.

Page forms derive from *z3c.form.form.Form*, which is extended by *plone.directives.form.Form* and *plone.directives.form.SchemaForm* as described in this manual. They will typically have actions, and be registered as a view for some context. For a completely standalone form, the site root is often good choice.

```
from five import grok
from plone.supermodel import model
from plone.directives import form

from z3c.form import button, field
from Products.CMFCore.interfaces import ISiteRoot
```

```

class IMyForm(model.Schema):

    ...

class MyForm(form.SchemaForm):
    grok.name('my-form')
    grok.require('zope2.View')
    grok.context(ISiteRoot)

    schema = IMyForm
    ignoreContext = True

    label = _(u"My form")
    description = _(u"A sample form.")

    @button.buttonAndHandler(_(u'Ok'))
    def handleOk(self, action):
        data, errors = self.extractData()

        if errors:
            self.status = self.formErrorsMessage
            return

        ...

    @button.buttonAndHandler(_(u"Cancel"))
    def handleCancel(self, action):
        ...

```

A non-schema version would look like this:

```

from five import grok
from plone.supermodel import model
from plone.directives import form

from z3c.form import button, field
from Products.CMFCore.interfaces import ISiteRoot

class IMyForm(model.Schema):

    ...

class MyForm(form.Form):
    grok.name('my-form')
    grok.require('zope2.View')
    grok.context(ISiteRoot)

    fields = field.Fields(IMyForm)
    ignoreContext = True

    label = _(u"My form")
    description = _(u"A sample form.")

    @button.buttonAndHandler(_(u'Ok'))
    def handleOk(self, action):
        data, errors = self.extractData()

```

```
        if errors:
            self.status = self.formErrorsMessage
            return

        ...

    @button.buttonAndHandler(_(u"Cancel"))
    def handleCancel(self, action):
        ...
```

Many “standalone” page forms will set *ignoreContext* = *True*. If it is *False* (the default), the form will read the current value of each field from the context, by attempting to adapt it to the form schema, as described in the previous section.

Sometimes, we want to populate the form with initial values that are not attributes of the context (or an adapter thereof). *z3c.form* allows us to change the object from which the form’s data is read, by overriding the *getContent()* method. We can return another object that provides or is adaptable to the schema interface(s) associated with the form’s fields, but we can also return a dictionary with keys that match the names of the fields in the form schema. This is usually easier than creating an adapter on some arbitrary context (e.g. the site root) solely for the purpose of pre-populating form values. It also makes it easy to construct the form’s initial values dynamically.

```
from five import grok
from plone.supermodel import model
from plone.directives import form

from zope import schema

from z3c.form import button, field
from Products.CMFCore.interfaces import ISiteRoot

...

class IMyForm(model.Schema):

    foo = schema.TextLine(title=_(u"Foo"))
    bar = schema.TextLine(title=_(u"Bar"))

class MyForm(form.SchemaForm):
    grok.name('my-form')
    grok.require('zope2.View')
    grok.context(ISiteRoot)

    schema = IMyForm
    ignoreContext = True

    label = _(u"My form")
    description = _(u"A sample form.")

    def getContent(self):
        data = {}
        data['foo'] = u"Foo"
        data['bar'] = u"Bar"
        return data

    @button.buttonAndHandler(_(u'Ok'))
    def handleOk(self, action):
        data, errors = self.extractData()

        if errors:
```

```

        self.status = self.formErrorsMessage
        return

    ...

    @button.buttonAndHandler(_(u"Cancel"))
    def handleCancel(self, action):
        ...

```

Note how the fields in the *data* dictionary returned by *getContent()* correspond to the fields of the schema interface from which the form's fields are built. If we had fields from multiple interfaces (e.g. using the *additional_schemata* tuple), we would need to populate keys based on the fields from all interfaces.

Also note that the values in the dictionary must be valid for the fields. Here, we have used *TextLine* fields, which expect unicode string values. We would likely get an error if the value was a byte string or integer, say.

Add forms

Forms to create new content objects

An add form, as its name implies, is used to add content to a container. Add forms are usually registered as views on a container. For generic CMF or Plone content, the *IFolderish* interface is normally a good candidate. The fields in an add form usually represent the fields in the type that is being added.

Note: If you are using Dexterity or Archetypes, these frameworks have their own add form factories, which you probably want to use instead of the more basic version described here.

Add forms derive from *z3c.form.form.AddForm*, which is extended by *plone.directives.form.AddForm* and *plone.directives.form.SchemaAddForm*, adding grok support and standard Plone semantics.

To use an add form, you must implement two methods - *create()* and *add()*. The form then takes care of emitting the proper events and directing the user to the newly added content item. You can also set the *immediate_view* property to the URL of a page to visit after adding the content item.

```

from five import grok
from plone.supermodel import model
from plone.directives import form

from z3c.form import button, field
from Products.CMFCore.interfaces import IFolderish

class IMyType(model.Schema):

    ...

class MyAddForm(form.SchemaAddForm):
    grok.name('add-my-type')
    grok.require('cmf.AddPortalContent')
    grok.context(IFolderish)

    schema = IMyType

    label = _(u"Add my type of content")
    description = _(u"A sample form.")

```

```
def create(self, data):
    type = MyType()
    type.id = data['id']
    type.title = data['title']
    ...

    return type

def add(self, object):
    self.context._setObject(object.id, object)
```

create() is called after validation, and is passed a dictionary of marshalled form fields. It should construct and return the object being added. That object is then passed to *add()* (after an object-created event has been fired), which should add it, normally to *self.context* (the container).

A non-schema version would look like this:

```
from five import grok
from plone.supermodel import model
from plone.directives import form

from z3c.form import button, field
from Products.CMFCore.interfaces import IFolderish

class IMyType(model.Schema):
    ...

class MyAddForm(form.AddForm):
    grok.name('add-my-type')
    grok.require('cmf.AddPortalContent')
    grok.context(IFolderish)

    fields = field.Fields(IMyType)

    label = _(u"Add my type of content")
    description = _(u"A sample form.")

    def create(self, data):
        type = MyType()
        type.id = data['id']
        type.title = data['title']
        ...

        return type

    def add(self, object):
        self.context._setObject(object.id, object)
```

Edit forms

Forms that edit something

Edit forms, unsurprisingly, are used to edit content objects or other contexts. They derive from *z3c.form.form.EditForm*, which is extended by *plone.directives.form.EditForm* and *plone.directives.form.SchemaEditForm*, adding grok support and Plone semantics. The edit form takes care of firing object-modified events, and implements default save and cancel actions.

Note: As with add forms, frameworks like Archetypes and Dexterity provide their own default edit forms, which should use for editing content objects built with those frameworks.

The schema of an edit form is normally a content object schema, which normally also describes the context of the form view. That is, the edit form is normally a view on the object that is being edited

That said, we can implement `getContent()` to supply a different context. This would normally provide the schema interface, but it does not need to. As with any form, the context need only be adaptable to the interface(s) associated with its fields.

A simple edit form in a view called `@@edit` that edits a content object providing *IMyType* would look like this:

```
from five import grok
from plone.supermodel import model
from plone.directives import form

from z3c.form import button, field
from Products.CMFCore.interfaces import IFolderish

class IMyType(form.Schema):

    ...

class MyAddForm(form.SchemaEditForm):
    grok.name('edit')
    grok.require('cmf.ModifyPortalContent')
    grok.context(IMyType)

    schema = IMyType

    label = _(u"Edit my type")
    description = _(u"Make your changes below.")
```

There is no need to define any actions or implement any methods. The default save button handler will adapt the context to *IMyType* and then set each field in the interface with the submitted form values.

A non-schema example would look like:

```
from five import grok
from plone.supermodel import model
from plone.directives import form

from z3c.form import button, field
from Products.CMFCore.interfaces import IFolderish

class IMyType(model.Schema):

    ...

class MyAddForm(form.EditForm):
    grok.name('edit')
    grok.require('cmf.ModifyPortalContent')
    grok.context(IMyType)

    fields = field.Fields(IMyType)

    label = _(u"Edit my type")
```

```
description = _(u"Make your changes below.")
```

As a slightly ore interesting example, here is one adapted from *plone.app.registry*'s control panel form base class:

```
from five import grok
from plone.supermodel import model
from plone.directives import form

from zope.component import getUtility

from z3c.form import button, field
from Products.CMFCore.interfaces import ISiteRoot

from plone.registry.interfaces import IRegistry

class IMySettings(model.Schema):

    ...

class MyAddForm(form.EditForm):
    grok.name('edit')
    grok.require('zope2.View')
    grok.context(IMyType)

    fields = field.Fields(IMyType)

    label = _(u"Edit my type")
    description = _(u"Make your changes below.")

class EditSettings(form.SchemaEditForm):
    grok.name('edit-my-settings')
    grok.require('cmf.ManagePortal')
    grok.context(ISiteRoot)

    schema = IMySettings

    label = _(u"Edit settings")

    def getContent(self):
        return getUtility(IRegistry).forInterface(self.schema)
```

The idea here is that *IMySettings*, which is set as the schema for this schema edit form, is installed in the registry as a set of records. The *forInterface()* method on the *IRegistry* utility returns a so-called records proxy object, which implements the interface, but reads/writes values from/to the configuration registry. The form view is registered on the site root, but we override *getContent()* to return the records proxy. Hence, the initial form values is read from the proxy, and when the form is successfully submitted, the proxy (and hence the registry) is automatically updated.

Display forms

Using widgets in display mode

Both forms and widgets support the concept of a “mode”. The form’s mode acts as a default for its widgets. The most commonly used mode is ‘input’, as indicated by the constant *z3c.form.interfaces.INPUT_MODE*, but there is also ‘hidden’ (*HIDDEN_MODE*) and ‘display’ (*DISPLAY_MODE*). The latter is the form mode for *display forms*.

Display forms derive from *z3c.form.display.DisplayForm*, which is extended by *plone.directives.dexterity.DisplayForm*. This also mixes in *plone.autoform.view.WidgetsView*, which provides

various conveniences for dealing with display mode widgets and fieldsets (groups). Note that this is a “schema form”, i.e. we must set the *schema* property (and optionally *additional_schemata*) to a schema deriving from *form.Schema*.

Note: If you require a grokked alternative that is not a schema form, you can derive from *z3c.form.form.DisplayForm* and *plone.directives.form.form.GrokkedForm*.

Display forms are not very common outside framework code. In most cases, it is easier to just create a standard view that renders the context. In a framework such as Dexterity, display forms are used as the default views of content items. The main reason to use display forms for anything bespoke is to use a complex widget that has a display mode rendering that is difficult to replicate in a custom template.

It is also possible to put some widgets into *input* mode (by setting the *mode* attribute in the *updateWidgets()* hook), thus placing a widget into a form that is otherwise not managed by *z3c.form*.

Display forms are used much like standard views. For example:

```
from five import grok
from plone.supermodel import model
from plone.directives import dexterity, form

...

class IMyContent(model.Schema):
    ...

class MyDisplayForm(dexterity.DisplayForm):
    grok.name('view')
    grok.require('zope2.View')
    grok.context(IMyContent)
```

There would typically also be a template associated with this class. This uses standard five.grok view semantics. If the display form above was in a module called *display.py*, a template may be found in *display_templates/mydisplayform.pt*.

The *DisplayForm* base class in *plone.directives.form* makes the following view attributes available to the template:

- *view.w* is a dictionary of all the display widgets, keyed by field names. This includes widgets from alternative fieldsets.
- *view.widgets* contains a list of widgets in schema order for the default fieldset.
- *view.groups* contains a list of fieldsets in fieldset order.
- *view.fieldsets* contains a dict mapping fieldset name to fieldset
- On a fieldset (group), you can access a *widgets* list to get widgets in that fieldset

The *w* dict is the mostly commonly used. To render a widget named *foo* in the template, we could do:

```
<span tal:replace="structure view/w/foo/render" />
```

Customising form presentation

Layout templates

Creating a custom layout for our form

To this point, we have relied on Plone (in fact, *plone.app.z3cform*) to supply a default template for our forms. This uses the default Plone form markup, which is consistent with that used in other forms in Plone. For many forms, this is all we need. However, it is sometimes useful to create a custom template.

Custom templates are normally needed for one of two reasons: Either, to insert some additional markup around or inside the form itself; or to radically change the form markup itself. The former is more common, since changing the form look-and-feel is normally done better with CSS. For that reason, *plone.app.z3cform* registers a view called `@@ploneform-macros`, which provides useful macros for rendering forms using the standard markup. We will illustrate how to use this below.

The easiest way to associate a template with a form is to use the default grokked template association. Our form is called *OrderForm* and lives in a module called *order.py*, so the grokker will look for a template in *order_templates/orderform.pt*.

Note: With the exception of *DisplayForms*, there is always a default template for forms extending the grokked base classes in *plone.directives.form*. Therefore, the template is optional. Unlike *grok.View* views, there is no need to override *render()* if the template is omitted.

As an example, let's create such a template and add some content before the form tag:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:metal="http://xml.zope.org/namespaces/metal"
      xmlns:tal="http://xml.zope.org/namespaces/tal"
      xmlns:i18n="http://xml.zope.org/namespaces/i18n"
      i18n:domain="example.dexterityforms"
      metal:use-macro="context/main_template/macros/master">

  <metal:block fill-slot="main">

    <h1 class="documentFirstHeading" tal:content="view/label | nothing" />

    <p>Welcome to Backgammon Pizza! We hope you enjoy our food.</p>

    <div id="content-core">
      <metal:block use-macro="context/@@ploneform-macros/titlelessform" />
    </div>

  </metal:block>

</html>
```

Notice how the `@@ploneform-macros` view does most of the work. This contains a number of useful macros:

- *form* is a full page form, including the label
- *titlelessform* includes the form *status* at the top, the `<form />` element, and the contents of the *fields* and *actions* macros. It also defines three slots: *formtop*, just inside the `<form>` opening tag; *formbottom*, just before the `</form>` closing tag; and *beforeactions*, just before the form actions (buttons) are output.
- *fields* iterates over all widgets in the form and renders each, using the contents of the *field* macro.
- *field* renders a single field. It expects the variable *widget* to be defined in the TAL scope, referring to a *z3c.form* widget instance. It will output an error message if there is a field validation error, a label, a marker to say whether the field is required, the field description, and the widget itself (normally just an `<input />` element).
- *actions* renders all actions on the form. This normally results in a row of `<input type="submit" ... />` elements.

Note: If you require more control, you can always create your form from scratch. Take a look at *macros.pt* in

plone.app.z3cform for inspiration.

If you don't require tabbed fieldsets or "inline" field validation, the template can be simplified substantially. See *macros.pt* in *plone.z3cform* for a cleaner example.

The most important variables used in the template are:

- *view.id*, a unique id for the form
- *view enctype*, the form's *enctype* attribute
- *view.label*, the form's title
- *view.description*, the forms' description
- *view.status*, a status message that is often set in action handlers.
- *view.groups*, a list of fieldsets (groups), as represented by *Group* instances.
- *view.widgets*, which contains all widgets. *view.widgets.errors* contains a list of error snippet views. Otherwise, *widgets* behaves like an ordered dictionary. Iterating over its *values()* will yield all widgets in order. The widgets have been updated, and can be output using their *render()* method.
- *view.actions*, contains an ordered dictionary of actions (buttons). Iterating over its *values()* will yield all actions in order. The actions have been updated, and can be output using their *render()* method.

Error snippets

Customising error messages

When creating custom validators, as shown earlier in this manual, it is easy to tailor an error message. However, *zope.schema* and *z3c.form* already perform a fair amount of validation for us, which results in generic error messages. For example, if a required field is not completed, a rather bland error message ("Required input is missing") will be shown. Sometimes, we may want to change these messages.

z3c.form allows error messages to be customised at various levels of detail. For example, it is possible to register a custom *z3c.form.interfaces.IErrorViewSnippet* adapter, which behaves like a mini-view and can output arbitrary HTML. However, in most cases, we only want to update the output text string. For this, we use what's known as a "value adapter". This is simply an adapter which *z3c.form*'s default *IErrorViewSnippet* implementations will look up to determine which message to show.

The easiest way to create an error message value adapter is to use the *@form.error_message()* decorator from *plone.directives.form*. This decorator should be applied to a function that takes as its only argument the (invalid) value that was submitted, and return a unicode string or message indicating the error. To illustrate this, we will add a new function to *order.py*, just after the interface definition:

```
from five import grok
from plone.supermodel import model
from plone.directives import form

from zope.interface import invariant, Invalid
from zope.component import queryUtility

from zope import schema

from zope.schema.interfaces import IContextSourceBinder
from zope.schema.interfaces import RequiredMissing
from zope.schema.vocabulary import SimpleVocabulary

...
```

```
from example.dexterityforms.interfaces import MessageFactory as _

...

class IPizzaOrder(model.Schema):

    ...

    telephone = schema.ASCIILine(
        title=_(u"Telephone number"),
        description=_(u"We prefer a mobile number"),
    )

    ...

@form.error_message(field=IPizzaOrder['telephone'], error=RequiredMissing)
def telephoneOmittedErrorMessage(value):
    return u"Without your telephone number, we can't contact you in case of a problem."
    ↪ "
```

As with the `@form.validator()` decorator, the `@form.error_message()` validator takes a number of keyword arguments, used to control where the error message is applied. The allowable arguments are:

error The type of error, which is normally represented by an exception class. The most general type will usually be a `zope.schema.interfaces.ValidationError`. See below for a list of other common exception types.

request The current request. This is normally used to supply a browser layer marker interface. This is a good way to ensure a general error message is only in force when our product is installed.

widget The widget which was used to render the field.

field The field to which the error message applies. If this is omitted, the message would apply to all fields on the form (provided *form* is supplied) of the given error (provided *error* is applied).

form The form class. We can use this either to apply a single message to a given error across multiple fields in one form (in which case *field* would be omitted), or to customise an error message for a particular form only if a schema is used in more than one form.

content The content item (context) on which the form is being rendered.

Note: In almost all cases, you will want to supply both *field* and *error* at a minimum, although if you have multiple fields that may raise a particular error, and you want to create a message for all instances of that error, you can omit *field* and use *form* instead. If you supply just *error*, the validator will apply to all instances of that error, on all forms, site-wide, which is probably not a good idea if you intend your code to be usable. At the very least, you should use the *request* field to specify a browser layer in this case, and install that layer with *browserlayer.xml* in your product's installation profile.

The exception types which may be used for the *error* discriminator are field-specific. The standard fields as defined in *zope.schema* use the following exceptions, all of which can be imported from *zope.schema.interfaces*:

- *RequiredMissing*, used when a required field is submitted without a value
- *WrongType*, used when a field is passed a value of an invalid type
- *TooBig* and *TooSmall*, used when a value is outside the *min* and/or *max* range specified for ordered fields (e.g. numeric or date fields)

- *TooLong* and *TooShort*, used when a value is outside the *min_length* and/or *max_length* range specified for length-aware fields (e.g. text or sequence fields)
- *InvalidValue*, used when a value is invalid, e.g. a non-ASCII character passed to an ASCII field
- *ConstraintNotSatisfied*, used when a *constraint* method returns *False*
- *WrongContainedType*, used if an object of an invalid type is added to a sequence (i.e. the type does not conform to the field's *value_type*)
- *NotUnique*, used if a uniqueness constraint is violated
- *InvalidURI*, used for *URI* fields if the value is not a valid URI
- *InvalidId*, used for *Id* fields if the value is not a valid id
- *InvalidDottedName*, used for *DottedName* fields if the value is not a valid dotted name

Further reading

Where to find more information

To find out more about *z3c.form* and its uses in Plone, consult the following references:

- The [z3c.form](#) documentation. This provides a detailed guide to *z3c.form*'s inner workings.
- The [plone.z3cform](#) documentation. Describes how to use “raw” *z3c.form* forms in Zope 2, and documents the Zope 2-specific extensions provided by this package.
- The [plone.app.z3cform](#) documentation. Describes how to use “raw” *z3c.form* forms in Plone.
- The [plone.autoform](#) documentation. Explains the raw behaviour of the *plone.autoform* library and its directives.
- The [plone.directives.form](#) documentation. Lists the form base classes.
- The Dexterity manual. Illustrates in various sections how *z3c.form* is used in Dexterity.

Training

A number of Plone trainers have joined forces to create completely open [Training materials](#).

While following a real-life course is the best way to get up to speed with Plone, the material is also very useful for self-study. You will find separate chapters on creating packages, writing your own theme and much more here.

Programming with Plone

Getting started

How to get started with Plone development.

Introduction

Plone is developed in the *Python* programming language. *You should master Python basics* before you can efficiently customize Plone. If you are very new to Python, Plone or software development, it is suggested that you read the [Professional Plone 4 Development book](#) before you attempt to develop your own solutions.

Plone runs on the top of the Zope 2 application server, meaning that one Zope 2 server process can contain and host several Plone sites. Plone also uses Zope 3 components. Zope 3 is not an upgrade for Zope 2, but a separate project.

Internally, Plone uses the objected-oriented *ZODB* database and the development mindset greatly differs from that of SQL based systems. SQL backends can still be integrated with Plone, like for any other Python application, but this is a more advanced topic.

Training

A number of Plone trainers have joined forces to create completely open [Training materials](#).

While following a real-life course is the best way to get up to speed with Plone, the material is also very useful for self-study. You will find separate chapters on creating packages, writing your own theme and much more here.

Installing Plone

It is recommended that you do Plone development on Linux or OS X. Development on Windows is possible, but you need to have much more experience dealing with Python and Windows related problems, so starting on Windows is not so easy.

See [installation instructions](#) for how to create a Plone installation suitable for development.

Non-programming approaches for customizing Plone

If you lack programming skill or resources, you can still get some things done in Plone:

- [Plomino](#) is a powerful and flexible web-based application builder for Plone
- [PloneFormGen](#) allows you to build forms in a web browser
- Plone comes with through-the-web Dexterity content type editor

However, for heavy customization, Python, JavaScript, TAL page templates and CSS programming is needed.

Enabling debug mode

By default, Plone runs in a *production mode* where changed files in the file system are not reflected in the served HTML. When you start developing Plone you need to first [put it into a debug mode](#).

Plone add-ons as Python packages

Plone sites can be customized by installing *Plone add-ons*, which add or customize functionality. You can install existing add-ons that others have developed or you can develop and install your own add-ons. Add-ons are developed and distributed as [Python packages](#). Many open-source Python packages, including Plone add-ons, are available from [PyPI \(the Python Package index\)](#).

Plone uses a tool called [Buildout](#) to manage the set of Python packages that are part of your Plone installation. Using Buildout involves using the `buildout.cfg` configuration file and the `bin/buildout` command.

Finding and installing add-on packages

Plone add-ons can be found at the [plone.org Products](#) page or at the [PyPI \(the Python Package index\)](#).

See the [Installing add-on packages using buildout](#) section for more details.

Creating your first add-on

Since Python egg package structure is little bit complex, to get started with your first add-on you can create a code skeleton (scaffold) for it using *bobtemplates for Plone*.

- Mr.Bob with the bobtemplates.plone generates a basic Python egg package with some Plone files in-place.
- This package is registered to buildout as a development egg in the `buildout.cfg` file.
- Buildout is rerun which regenerates your `bin/instance` script with the new set of Python eggs.
- You start your Plone instance in debug mode.
- You install your add-on through `Add/remove add-ons`

If you want to create a package with Dexterity content types please read about Setting up a Dexterity project.

Plone development workflow

You never edit Plone files directly. Everything under `parts` and `eggs` folders in your Plone installation is downloaded from the Internet and dynamically generated by Buildout, based on `buildout.cfg`. Buildout is free to override these files on any update.

You need to have your own add-on in the `src/` folder as created above. There you overlay changes to the existing Plone core through extension mechanisms provided by Plone:

- *Layers*
- *Adapters*
- *Installation profiles*

Plone development always happens on your local computer or the development server. The changes are moved to production through version control system like Git or Subversion.

The best practice is that you install Plone on your local computer for development.

Plone add-on features

Note: Explain that Archetypes are old and basically there to support upgraded sites, but that new development should use Dexterity, maybe remove them even ?

Plone add-ons usually:

- Create custom *content types* or extend existing ones for your specialized need. Plone has two subsystems for <content types: *Dexterity*.
- Add new *views* for your site and its content.
- Create Python-processed *forms* on your site.
- Theme your site
- etc.

A lot of Plone functionality is built on *Zope 3 development patterns* like adapters and interfaces. These design patterns take some time to learn, but they are crucial in complex component based software like Plone.

Development mode restarts

Plone must be started in the development mode using `bin/instance fg` command. Then

- JavaScript files are in debug mode and automatically loaded when you hit refresh
- CSS files are in debug mode and automatically loaded when you hit refresh
- TAL page templates (.pt files) are automatically reloaded on every request
- *GenericSetup XML files are reloaded*

Please note that Plone development mode does not reload `.py` or `.zcm1` files by default. This is possible, however. Use the [sauna.reload](#) package to make Plone reload your Python code automatically when it is changed.

Through-the-web customizations

Some aspects of Plone can be changed through the Management Interface. Documentation here does not focus on extending functionality through the Management Interface because this method is severely limited and usually can take you only half way there.

Plone resources

- [Plone Issue Tracker](#) contains bug reports, Plone source code and commits. Useful when you encounter a new exception or you are looking for a reference on how to use the API.
- [Plone source code in version control system.](#)
- [Plone API \(in development\).](#)

Zope resources

- [Zope source code in version control system.](#)
- [Zope 2 book.](#) This describes old Zope 2 technologies. The book is mostly good for explaining some old things, but “do not” use it as a reference for building new things.

The chapters on Zope Page Templates however are still the best reference on the topic.

Python resources

Python for beginners

Description

The basics of Python programming, and performing Python interpreter installations.

Introduction

[Python](#) is the programming language used by [Plone](#) and [Zope](#). One needs to have at least basic Python experience before considering building Plone add-ons or customizations.

Note: You should not try to write programs for Plone before you can program Python on the basic level.

Python tutorials and online classes

- [Official Python tutorial](#)
- [Google Python classes](#)
- [Free Python books](#)
- [Dive into Python book](#)
- [Python at codecademy.org](#)

Debug mode explained

Debug mode

Description

Plone can be put in the debug mode where one can diagnose start up failures and any changes to CSS, JavaScript and page templates take effect immediately.

Introduction

By default when you start Plone you start it in a **production mode**.

- Plone is faster
- CSS and JavaScript files are *merged* instead of causing multiple HTTP request to load these assets. CSS and JavaScript behavior is different in production versus debug mode, especially with files with syntax errors because of merging.
- Plone does not reload changed files from the disk

Because of above optimizations the development against a production mode is not feasible. Instead you need to start Plone in debug mode (also known as development mode) if you are doing any site development.

In debug mode

- If Plone start-up fails, the Python traceback of the error is printed in the terminal
- All logs and debug messages are printed in the terminal; Zope process does not detach from the terminal
- Plone is slower
- CSS and JavaScript files are read file-by-file so line numbers match on the actual files on the disk. (*portal_css* and *portal_javascript* is set to debug mode when Plone is started in debug mode)
- Plone reloads CSS, JavaScript and .pt files when the page is refreshed

Note: Plone does not reload .py or .zcmf files in the debug mode by default.

Reloading Python code

Reloading Python code automatically can be enabled with [sauna.reload](#) add-on.

JavaScript and CSS issues with production mode

See **portal_css** and **portal_javascript** in the Management Interface to inspect how your scripts are bundled.

Make sure your JavaScript and CSS files are valid, mergeable and compressable. If they are not then you can tweak the settings for individual file in the corresponding management tool.

Refresh issues

Plone **production mode** should re-read CSS and JavaScript files on Plone start-up.

Possible things to debug and force refresh of static assets

- Check HTML <head> links and the actual file contents
- Go to *portal_css*, press *Save* to force CSS rebundling
- Make sure you are not using *plone.app.caching* and doing caching forever
- Use [hard browser refresh](#) to override local cache

Starting Plone in debug mode on Microsoft Windows

This document explains how to start and run the latest Plone (Plone 4.1.4) on Windows 7. This document explains post-installer steps on how to start and enter into a Plone site. Installation

Installation

This quick start has been tested on Windows 7. Installation remains the same on older versions of Windows through WinXP.

1. Run installer from [Plone.org](#) download page
2. The Plone buildout directory will be installed in C:\Plone41
3. The installer will launch your Plone instance when it finishes. To connect, direct your browser to: <http://localhost:8080>

Note: In the buildout bin directory you'll find the executable files to control Plone instance.

Starting and Stopping Plone

If your Plone instance is shutdown you can start and control it from the command prompt.

Note: To control Plone you need to execute your command prompt as an administrator.

In the command prompt enter the following command to access your buildout directory (the varies according to Plone version):

```
cd "C:\\Plone41"
```

To start Plone in debug mode type:

```
bin\\instance fg
```

You can interrupt the instance by pressing CTRL-C. This will also take down the Zope application server and your Plone site.

Accessing Plone

When you launch Plone in debug or daemon mode it will take a few moments to launch. If you are in debug mode, Plone will be ready serve pages when the following line is displayed in your command prompt:

```
INFO Zope Ready to handle requests
```

When the instance is running and listing to port 8080, point your browser to address on your local computer:

```
http://localhost:8080
```

The Plone welcome screen will load and you can create your first Plone site directly by clicking the **Create a new Plone Site** button.

A form will load asking for the *Path Identifier* (aka the site id) and *Title* for a new Plone site. It will also allow you to select the main site language, and select any add-on products you wish to install with the site.

Note: These entries can all be modified once the site is created. Changing the site id is possible, but not recommended.

To create your site, fill in this form and click the *Create Plone Site* button. Plone will then create and load your site.

Note: The url of your local Plone instance will end with the site id you set when setting up your site. If the site id were *Plone* then the resultant URL is: *http://localhost:8080/Plone*.

Congratulations! You should be now logged in as an admin to your new Plone instance and you'll see the front page of Plone.

Starting Plone in debug mode on UNIX

Single instance installation ("zope")

Enter to your installation folder using `cd` command (depends on where you have installed Plone):

```
cd ~/Plone/zintance # Default local user installation location
```

For root installation the default location is `/usr/local/Plone`.

Type in command:

```
bin/instance fg
```

Press CTRL+C to stop.

Clustered installation (“zeo”)

If you have ZEO cluster mode installation you can start individual processes in debug mode:

```
cd ~/Plone/zeocluster
bin/zeoserver fg & # Start ZODB database server
bin/client1 fg & # Start ZEO front end client 1 (usually port 8080)
# bin/client2 fg # For debugging issues it is often enough to start client1
```

Determining programmatically whether Zope is in debug mode

Zope2’s shared global data *Globals*, keeps track on whether Zope2 is started in debug mode or not.:

```
import Globals
if Globals.DevelopmentMode:
    # Zope is in debug mode
```

Note: There is a difference between Zope being in debug mode and the JavaScript and CSS resource registries being in debug mode (although they will automatically be set to debug mode if you start Zope in debug mode).

HTTP serving and traversing site data

Serving content from your site to your users is effectively a mechanism to generate HTTP responses to HTTP requests.

In Plone, answering to HTTP requests can be divided to three subproblems:

- managing the lifecycle of the HTTP request and response pair;
- publishing, by traversing the request to the target object by its URI;
- choosing different parts of the code depending on active layers.

Plone and Zope 2 application servers support FTP, WebDAV and XML-RPC protocols besides plain HTTP.

HTTP request and response

Description

Accessing and manipulating Zope’s HTTP request and response objects programmatically.

Introduction

This chapter explains the basics of Zope HTTP requests and responses:

- request and response objects lifecycle;

- data which can be extracted from the request;
- data which can be placed on the response.

Lifecycle

Unlike some other web frameworks, in Plone you do not explicitly create or return HTTP response objects. A HTTP request object always has a HTTP response object associated with it, and the response object is created as soon as the request hits the webserver.

The response is available for the whole lifetime of request processing. This effectively allows you to set and modify response headers at any point in the code.

Webservers

Usually Plone runs on Zope's [ZServer](#) (based on Sam Rushing's [Medusa](#)). Other alternatives are [WSGI](#) compatible web servers like [Repoze](#).

The web server will affect how your HTTP objects are constructed.

HTTP Request

All incoming HTTP requests are wrapped in Zope's [ZPublisher HTTPRequest](#) objects. This is a multi-mapping: it contains mappings for environment variables, other variables, form data, and cookies, but the keys of all these mappings can also be looked up directly on the request object (i.e. `request['some_form_id']` and `request.form['some_form_id']` are equivalent).

Usually your view function or instance will receive an HTTP request object, along with a traversed context, as its construction parameter.

You can access the request in your view:

```
from Products.Five.browser import BrowserView

class SampleView(BrowserView):

    def __init__(self, context, request):
        # Each view instance receives context and request as construction parameters
        self.context = context
        self.request = request

    def __call__(self):
        # Entry point of request processing
        # Dump out incoming request variables
        print self.request.items()
```

Request method

The request method (GET or POST) can be read:

```
request["REQUEST_METHOD"] == "POST" # or "GET"
```

Request URL

To get the requested URL:

```
>>> request["ACTUAL_URL"]
'http://localhost:8080/site'
```

To get the URL of the served object use the following (this might be different from the requested URL, since Plone does all kinds of default page and default view magic):

```
>>> request["URL"]
'http://m.localhost:8080/site/matkailijallefolder/@@frontpage'
```

Note: URLs, as accessed above, do not contain query string.

Query string

The unparsed query string can be accessed.

E.g. if you go to `http://localhost:8080/site?something=foobar`:

```
>>> self.request["QUERY_STRING"]
'something=foobar'
```

If the query string is not present in the HTTP request, it is an empty string.

Note: You can also use the `request.form` dictionary to access parsed query string content.

Request path

The request URI path can be read from `request.path`, which returns a list of path components. `request.path` is a virtual path, and has the site id component removed from it.

Example:

```
reconstructed_path = "/".join(request.path)
```

Other possible headers:

```
('PATH_INFO', '/plonecommunity/Members')
('PATH_TRANSLATED', '/plonecommunity/Members')
```

REQUEST_URI

To get the variable which corresponds to `REQUEST_URI` in e.g. PHP the following helps:

```
# Concatenate the user-visible URL and query parameters
full_url = request.ACTUAL_URL + "?" + request.QUERY_STRING
parsed = urlparse.urlsplit(full_url)
```

```
# Extract path part and add the query if it existed
uri = parsed[2]
if parsed[3]:
    uri += "?" + parsed[3]
```

For more information, see:

- http://www.teamrubber.com/blog/_serverrequest_uri-in-zope/
- <http://www.doughellmann.com/PyMOTW/urlparse/index.html>

Request client IP

Example:

```
def get_ip(request):
    """ Extract the client IP address from the HTTP request in a proxy-compatible way.

    @return: IP address as a string or None if not available
    """
    if "HTTP_X_FORWARDED_FOR" in request.environ:
        # Virtual host
        ip = request.environ["HTTP_X_FORWARDED_FOR"]
    elif "HTTP_HOST" in request.environ:
        # Non-virtualhost
        ip = request.environ["REMOTE_ADDR"]
    else:
        # Unit test code?
        ip = None

    return ip
```

For functional tests based on `zope.testbrowser` use the `addHeader` method to add custom headers to a browser.

GET variables

HTTP GET variables are available in `request.form` if the `REQUEST_METHOD` was GET.

Example:

```
# http://yoursite.com/@testview/?my_param_id=something
print self.request.form["my_param_id"]
```

POST variables

HTTP POST variables are available in `request.form`:

```
print request.form.items() # Everything POST brought to us
```

There is no difference in accessing GET and POST variables.

Request body

The request body can be retrieved from the `HTTPRequest` object by using the `get` method with the key `BODY`:

```
print request.get('BODY') # Prints the content of the request body
```

HTTP headers

HTTP headers are available through `request.get_header()` and the `request.environ` dictionary.

Example:

```
referer = self.request.get_header("referer") # Page referer (the page from user came_
↳ from)

if referer == None: # referer will be none if it was missing
    pass
```

Dumping all headers:

```
for name, value in request.environ.items():
    print "%s: %s" % (name, value)
```

A Management Interface Python script to dump all HTTP request headers:

```
from StringIO import StringIO

# Import a standard function, and get the HTML request and response objects.
from Products.PythonScripts.standard import html_quote
request = container.REQUEST
response = request.response

buffer = StringIO()

response.setHeader("Content-type", "text/plain")

for name, value in request.environ.items():
    print >> buffer, "%s: %s" % (name, value)

return buffer.getvalue()
```

Query string

To access the raw HTTP GET query string:

```
query_string = request["QUERY_STRING"]
```

Web environment

The web server exposes its own environment variables in `request.other` (`ZServer`) or `request.environ` (`Repoze` and other `WSGI`-based web servers):


```
print request.other.items()

user_agent = request.other["HTTP_USER_AGENT"]

user_agent = request.environ["HTTP_USER_AGENT"] # WSGI or Repoze server
```

Hostname

Below is an example to get the HTTP server name in a safe manner, taking virtual hosting into account:

```
def get_hostname(request):
    """ Extract hostname in virtual-host-safe manner

    @param request: HTTPRequest object, assumed contains environ dictionary

    @return: Host DNS name, as requested by client. Lowercased, no port part.
             Return None if host name is not present in HTTP request headers
             (e.g. unit testing).

    """

    if "HTTP_X_FORWARDED_HOST" in request.environ:
        # Virtual host
        host = request.environ["HTTP_X_FORWARDED_HOST"]
    elif "HTTP_HOST" in request.environ:
        # Direct client request
        host = request.environ["HTTP_HOST"]
    else:
        return None

    # separate to domain name and port sections
    host=host.split(":")[0].lower()

    return host
```

See also

- http://httpd.apache.org/docs/2.1/mod/mod_proxy.html#x-headers
- <http://zotonic.googlecode.com/hg/doc/varnish.zotonic.vcl> (X-Forwarded-Host)

Request port

It is possible to extract the Zope instance port from the request. This is useful e.g. for debugging purposes if you have multiple ZEO front ends running, and you want to identify them:

```
port = request.get("SERVER_PORT", None)
```

Note: The SERVER_PORT variable returns the port number as a string, not an integer.

Note: This port number is not the one visible to the external traffic (port 80, HTTP)

Published object

`request["PUBLISHED"]` points to a view, method or template which was the last item in the traversing chain to be called to render the actual page.

To extract the relevant content item from this information you can do e.g. in the after publication hook:

```
def find_context(request):
    """Find the context from the request

    http://stackoverflow.com/questions/10489544/getting-published-content-item-out-of-
    requestpublished-in-plone
    """
    published = request.get('PUBLISHED', None)
    context = getattr(published, '__parent__', None)
    if context is None:
        context = request.PARENTS[0]
    return context
```

- You might also want to filter out CSS etc. requests
- Please note that `request["PUBLISHED"]` is set after language negotiation and authentication
- [More complete example](#)

Flat access

GET, POST and web environment variables are flat mapped to the request object as a dictionary look up:

```
# Comes from POST
request["input_username"] == request.form["input_username"]

# Comes from environ
request.get('HTTP_USER_AGENT') == request.environ["HTTP_USER_AGENT"]
```

Request mutability

Even if you can write and add your own attributes to HTTP request objects, this behavior is discouraged. If you need to create cache variables for request lifecycle use [annotations](#).

Accessing HTTP request outside context

There are often cases where you would like to get hold of the HTTP request object, but the underlying framework does not pass it to you. In these cases you have two ways to access the request object:

- Use *acquisition* to get the request object from the site root. When Plone site traversal starts, the HTTP request is assigned to current site object as the `site.REQUEST` attribute.
- Use <https://pypi.python.org/pypi/five.globalrequest>.

Example of getting the request using acquisition:

```
# context is any traversed Plone content item
request = getattr(context, "REQUEST", None)
if request is not None:
```

```

    # Do the normal flow
    ...
else:
    # This code path was not initiated by an incoming web server request
    # Handle cases like
    # - command line scripts
    # - add-on installer
    # - code called during Zope start up
    # -etc.
    ...

```

zope.globalrequest.getRequest

See

- <https://pypi.python.org/pypi/five.globalrequest>

HTTP response

Usually you do not return HTTP responses directly from your views. Instead, you modify the existing HTTP response object (associated with the request) and return the object which will be HTTP response payload.

The returned payload object can be:

- a string (str) 8-bit raw data; or
- an iterable: the response is streamed, instead of memory-buffered.

Accessing response

You can access the HTTP response if you know the request:

```

from Products.Five.browser import BrowserView

class SampleView(BrowserView):

    def __init__(context, request):
        # Each view instance receives context and request as construction parameters
        self.context = context
        self.request = request

    def __call__(self):
        response = self.request.response
        return "<html><body>Hello world!</body></html>"

```

Response headers

Use `HTTPResponse.setHeader()` to set headers:

```

# The response is a dynamically generated image
self.request.response.setHeader("Content-type", "image/jpeg")
return image_data

```

Content disposition

The Content-Disposition header is used to set the filename of a download. It is also used by Flash 10 to check whether Flash download is valid.

Example of setting the download and downloadable filename:

```
response = self.request.response
response.setHeader("Content-type", "text/x-vCard; charset=utf-8")
response.setHeader("Content-Transfer-Encoding", "8bit")

cd = 'attachment; filename=%s.vcf' % (context.id)
response.setHeader('Content-Disposition', cd)
```

For more information, see:

- <http://www.littled.net/new/2008/10/17/plone-and-flash-player-10/>
- <http://support.microsoft.com/kb/260519>

Return code

Use `HTTPResponse.setStatus(self, status, reason=None, lock=None)` to set HTTP return status (“404 Not Found”, “500 Internal Error”, etc.).

If `lock=True`, no further modification of the `HTTPResponse` status are allowed, and will fail silently.

Response body

You might want to read or manipulate the response body in the post-publication hook.

The response body is not always a string or basestring: it can be a generator or iterable for blob data.

The body is available as the `response.body` attribute.

You can change the body using `setBody` and locking it:

```
#lets empty the body and lock it
response.setBody('', lock=True)
```

If `lock=True`, no further modification of the `HTTPResponse` body are allowed, and will fail silently.

Redirects

Real redirects

Use the `response.redirect()` method:

```
# This will send a "302 Temporary Redirect" notification to the browser
response.redirect(new_url)

# This will send a "301 Permanent Redirect" notification to the browser
response.redirect(new_url, status=301)
```

You can lock the status to not let other change the status later in the process

```
response.redirect(new_url, lock=True)
```

JavaScript redirects

You can invoke this JavaScript redirect trick from a page template head slot in a hacky way

Cookies

See *cookies documentation*.

Middleware-like hooks

Plone does not have a middleware concept, as everything happens through traversal. Middleware behavior can be emulated with the *before traverse* hook. This hook can be installed on any persistent object in the traversing graph. The hook is persistent, so it is a database change and must be installed using custom GenericSetup Python code.

Warning: Before traverse hooks cannot create new HTTP responses, or return alternative HTTP responses. Only exception-like HTTP response modification is supported, e.g. HTTP redirects. If you need to rewrite the whole response, the post-publication hook must be used.

For more information, see:

- http://blog.fourdigits.nl/changing-your-plone-theme-skin-based-on-the-objects-portal_type
- http://zebert.blogspot.com/2008_01_01_archive.html
- <http://svn.repoze.org/thirdparty/zopelib/branches/2.9.8/ZPublisher/tests/testBeforeTraverse.py>

Examples:

- Redirector: <https://plonegomobile.googlecode.com/svn/trunk/gomobile/gomobile.mobile/gomobile/mobile/postpublication.py>

Transform chain

Transform chain is a hook into repoze.zope2 that allows third party packages to register a sequence of hooks that will be allowed to modify the response before it is returned to the browser.

It is used e.g. by `plone.app.caching`.

More information

- <https://pypi.python.org/pypi/plone.transformchain>

Post-publication hook

The post-publication hook is run when:

- the context object has been traversed;
- after the view has been called and the view has rendered the response;
- before the response is sent to the browser;
- before the transaction is committed.

This is practical for caching purposes: it is the ideal place to determine and insert caching headers into the response. Read more at the [plone.postpublicationhook package](#) page.

Custom redirect mappings

Below is an example how to install an event handler which checks in the site root for a TTW Python script and if such exist it asks it to provide a HTTP redirect.

This behavior allows you to write site-wide redirects

- In Python (thank god no Apache regular expressions)
- Redirects can access Plone content items
- You can have some redirects migrated from the old (non-Plone) sites

Add the event subscriber to `configure.zcml`:

```
<configure
  xmlns="http://namespaces.zope.org/zope"
  xmlns:browser="http://namespaces.zope.org/browser"
  xmlns:plone="http://namespaces.plone.org/plone"
  i18n_domain="example.dexterityforms">

  ...

  <subscriber
    for="zope.traversing.interfaces.IBeforeTraverseEvent"
    handler=".redirect.check_redirect"
  />

</configure>
```

Create a file `redirect.py` and add the code below. Remember to add `url` to *Parameter list* of the script on the script edit view:

```
"""
    Call a custom TTW script and allow it to handle redirects.

    Use the Management Interface to add a ``Script (Python)`` item named ``redirect_
    ↪handler``
    to your site root - you can edit this script in fly to change the redirects.

    * Redirect script must contain ``url`` in its parameter list
"""

import logging
from zope.component.hooks import getSite
from zExceptions import Redirect
from Products.CMFCore.interfaces import ISiteRoot

logger = logging.getLogger("redirect")

def check_redirect(event):
    """
```

Check if we have a custom redirect script in Zope application server root.

If we do then call it and see if we get a redirect.

The script itself is TTW Python script which may return string in the case of redirect or None if no redirect is needed.

For more examples, check

```
https://github.com/zopefoundation/Zope/blob/master/src/Zope2/App/tests/
→testExceptionHandler.py
"""
site = getSite()
request = event.request

url = request["ACTUAL_URL"]

if "no_redirect" in request.form:
    # Use no_redirect query parameter to disable this behavior in the case
    # you mess up with the redirect script
    return

# Check if we have a redirect handler script in the site root
if "redirect_handler" in site:

    try:
        # Call the script and get its output
        value = site.redirect_handler(url)
    except Exception, e:
        # No silent exceptions plz
        logger.error("Redirect exception for URL:" + url)
        logger.exception(e)
        return

    if value is not None and value.startswith("http"):
        # Trigger redirect, but only if the output value looks sane
        raise Redirect, value
```

Then an example `redirect_handler` script added through the Management Interface. Remember to add `url` to the *Parameter List* field of TTW (through the web) interface:

```
if "blaablaa" in url:
    return "http://webandmobile.mfabrik.com"
```

Or more complex example:

```
# Don't leak non-themed interface fom port 80
if ("manage.") in url and (not "8080" in url):
    return "http://manage.underconstruction.mfabrik.com:8080/LS"

if url == "http://underconstruction.mfabrik.com/":
    return "http://underconstruction.mfabrik.com/special-front-page"

# Redirect to the actual front page
if url == "http://site.com/":
    return "http://www.site.com/special-front-page"

if url == "http://www.site.com/":
```

```
    return "http://www.site.com/special-front-page"

if url.startswith("http://underconstruction.mfabrik.com/"):
    return url.replace("underconstruction.mfabrik.com", "www.site.com")

# Make sure that search engines and visitors access the site only using www. prefix
if url.startswith("http://site.com/"):
    return url.replace("site.com", "www.site.com")
```

Extracting useful information in the post-publication hook

Example:

```
from zope.component import adapter, getUtility, getMultiAdapter
from plone.postpublicationhook.interfaces import IAfterPublicationEvent
from Products.CMFCore.interfaces import IContentish

def get_contentish(object):
    """
    Traverse acquisition upwards until we get contentish object used for the HTTP_
    ↪response.
    """
    contentish = object
    while not IContentish.providedBy(contentish):
        if hasattr(contentish, "aq_parent"):
            contentish = contentish.aq_parent
        else:
            break
    return contentish

# This must be referred in configure.zcml
@adapter(Interface, IAfterPublicationEvent)
def language_fixer(object, event):
    """ Redirect mobile users to mobile site using gomobile.mobile.interfaces.
    ↪IRedirector.

    Note: Plone does not provide a good hook doing this before traversing, so we must
    do it in post-publication. This adds extra latency, but is doable.
    """
    request = event.request
    response = request.response

    # object can be a page template, view, whichever happens to be at the very end of_
    ↪traversed acquisition chain
    context = get_contentish(object)
```

Cross-origin resource sharing (CORS)

- <http://enable-cors.org/>
- https://developer.mozilla.org/En/HTTP_access_control

Traversing

Description

Plone content is organized as a tree. Traversing means looking up content from this tree by path. When HTTP request hits a Plone server, Plone will traverse the corresponding content item and its view function by URI.

Introduction

In Plone, all content is mapped to a single tree: content objects, user objects, templates, etc. Even most object methods are directly mapped to HTTP-accessible URIs.

Each object has a path depending on its location. *Traversal* is a method of getting a handle on a persistent object in the ZODB object graph from its path.

Traversal can happen in two places:

- When an HTTP request hits the server, the method on the object which will generate the HTTP response is looked up using traversal.
- You can manually traverse the ZODB tree in your code to locate objects by their path.

When an HTTP request is being published the traversing happens in `ZPublisher.BaseRequest.traverse`

- <https://github.com/zopefoundation/Zope/blob/master/src/ZPublisher/BaseRequest.py>

... but Zope includes other traversers, like `unrestrictedTraverse()` in the OFS module. Different traversing methods behave differently and may fire different events.

Object ids

Each content object has an id string which identifies the object in the parent container. The id string is visible in the browser address bar when you view the object. Ids are also visible in the Management Interface.

Besides id strings, the content objects have Unique Identifiers, or **UID**, which do not change even if the object is moved or renamed.

Though it's technically possible for ids to contain spaces or slashes, this is seldom a good idea, as it complicates working with ids in various situations.

Path

The Zope *path* is the location of the object in the object graph. It is a sequence of id components from the parent node(s) to the child separated by slashes.

Note: A path need not always be a sequence of object ids. During traversal, an object may consume subsequent path elements, interpreting them however it likes.

Example:

```
documentation/howTos/myHowTo
```

Exploring Zope application server

You can use the Management Interface to explore the content of your Zope application server:

- Sites
- Folders within the sites
- ...and so on

The Management Interface does not expose individual attributes. It only exposes traversable content objects.

Attribute traversing

Zope exposes child objects as attributes.

Example:

```
# you have obtained the plone.org portal root object somehow and it's
# stored in local variable "portal"

documentation = portal.documentation
howTos = getattr(portal, "how-to") # note that we need use getattr because dash is
→invalid in syntax
myHowTo = getattr(howTos, "manipulating-plone-objects-programmatically")
```

Container traversing

Zope exposes child objects as container accessor.

Example:

```
# you have obtained the plone.org portal root object somehow and it's
# stored in a local variable "portal"

documentation = portal["documentation"]
howTos = documentation["how-to"]
myHowTo = howTos["manipulating-plone-objects-programmatically"]
```

Traversing by full path

Any content object provides the methods `restrictedTraverse()` and `unrestrictedTraverse()`. See [Traversable](#).

Security warning: `restrictedTraverse()` executes with the privileges of the currently logged-in user. An `Unauthorized` exception is raised if the code tries to access an object for which the user lacks the *Access contents information* and *View* permissions.

Example:

```
myHowTo = portal.restrictedTraverse("documentation/howTos/myHowTo")

# Bypass security
myHowTo = portal.unrestrictedTraverse("documentation/howTos/myHowTo")
```

Warning: `restrictedTraverse()/unrestrictedTraverse()` does not honor `IPublishTraverse` adapters. [Read more about the issue in this discussion.](#)

Getting the object path

An object has two paths:

- The *physical path* is the absolute location in the current ZODB object graph. This includes the site instance name as part of it.
- The *virtual path* is the object location relative to the Plone site root.

Path mangling warning: Always store paths as virtual paths, or persistently stored paths will corrupt if you rename your site instance.

See [Traversable](#).

Getting physical path

Use `getPhysicalPath()`. Example:

```
path = portal.getPhysicalPath() # returns "plone"
```

Getting virtual path

For content items you can use `absolute_url_path()` from [OFS.Traversable](#):

```
path = context.absolute_url_path()
```

Map physical path to virtual path using HTTP request object `physicalPathToVirtualPath()`. Example:

```
request = self.request # HTTPRequest object

path = portal.document.getPhysicalPath()

virtual_path = request.physicalPathToVirtualPath(path) # returns "document"
```

Note: The virtual path is not necessarily the path relative to the site root, depending on the virtual host configuration.

Getting item path relative to the site root

There is no a direct, easy way to accomplish this.

Example:

```
from zope.component import getMultiAdapter

def getSiteRootRelativePath(context, request):
    """ Get site root relative path to an item
```

```
@param context: Content item which path is resolved

@param request: HTTP request object

@return: Path to the context object, relative to site root, prefixed with a slash.
"""

portal_state = getMultiAdapter((context, request), name=u'plone_portal_state')
site = portal_state.portal()

# Both of these are tuples
site_path = site.getPhysicalPath();
context_path = context.getPhysicalPath()

relative_path = context_path[len(site_path):]

return "/" + "/".join(relative_path)
```

Getting canonical object (breadcrumbs, visual path)

The visual path is presented in the breadcrumbs. It is how the site visitor sees the object path.

It may differ from the physical path:

- The *default content item* is not shown in the visual path.
- The *default view* is not shown in the visual path.

The canonical object is the context object which the user sees from the request URL:

Example:

```
context_helper = getMultiAdapter((context, self.request), name="plone_context_state")
canonical = context_helper.canonical_object()
```

Getting object URL

Use `absolute_url()`. See [Traversable](#).

URL mangling warning: `absolute_url()` is sensitive to virtual host URL mappings. `absolute_url()` will return different results depending on if you access your site from URLs `http://yourhost/` or `http://yourhost:8080/Plone`. Do not persistently store the result of `absolute_url()`.

Example:

```
url = portal.absolute_url() # http://nohost/plone in unit tests
```

Getting the parent

The object *parent* is accessible is `acquisition` chain for the object is set.

Use `aq_parent`:

```
parent = object.aq_parent
```

The parent is defined as `__parent__` attribute of the object instance:

```
object.__parent__ = object.aq_parent
```

`__parent__` is set when object's `__of__()` method is called:

```
view = MyBrowserView(context, request)
view = view.__of__(context) # Inserts view into acquisition chain and acquisition_
↪ functions become available
```

Getting all parents

Example:

```
def getAcquisitionChain(object):
    """
    @return: List of objects from context, its parents to the portal root

    Example::

        chain = getAcquisitionChain(self.context)
        print "I will look up objects:" + str(list(chain))

    @param object: Any content object
    @return: Iterable of all parents from the direct parent to the site root
    """

    # It is important to use inner to bootstrap the traverse,
    # or otherwise we might get surprising parents
    # E.g. the context of the view has the view as the parent
    # unless inner is used
    inner = object.aq_inner

    iter = inner

    while iter is not None:
        yield iter

        if ISiteRoot.providedBy(iter):
            break

        if not hasattr(iter, "aq_parent"):
            raise RuntimeError("Parent traversing interrupted by object: " +
                               ↪ str(iter))

        iter = iter.aq_parent
```

Getting the site root

You can resolve the site root if you have the handle to any context object.

Using portal_url tool

Example:

```
from Products.CMFCore.utils import getToolByName

# you know some object which is referred as "context"
portal_url = getToolByName(context, "portal_url")
portal = portal_url.getPortalObject()
```

You can also do shortcut using acquisition:

```
portal = context.portal_url.getPortalObject()
```

Note: Application code should use the `getToolByName` method, rather than simply acquiring the tool by name, to ease forward migration (e.g., to Zope3).

Using `getSite()`

Site is also stored as a thread-local variable. In Zope each request is processed in its own thread. Site thread local is set when the request processing starts.

You can use this method even if you do not have the context object available, assuming that your code is called after Zope has traversed the context object once.

Example:

```
from zope.component.hooks import getSite

site = getSite() # returns portal root from thread local storage
```

Note: Before Plone 4.3 `getSite` resided in `zope.app.component.hooks`. See <https://plone.org/documentation/manual/upgrade-guide/version/upgrading-plone-4.2-to-4.3/referencemanual-all-pages>

Note: Due to the fact that Plone does not show the default content item as a separate object, the page you are viewing in the browser from the site root URL is not necessary the root item itself. For example, in the default Plone installation this URL internally maps to Page whose id is `front-page` and you can still query the actual parent object which is the site root.

If you need to traverse using user visible breadcrumbs, see how breadcrumbs viewlet code does it.

Traversing back to the site root

Sometimes `getSite()` or `portal_url` are not available, but you still have the acquisition chain intact. In these cases you can simply traverse parent objects back to the site root by iterating over the acquisition-chain or using the `aq_parent` accessor:

```
from Products.CMFCore.interfaces import ISiteRoot

def getSite(context):

    if not ISiteRoot.providedBy(context):
        return context
```

```

else:
    for item in context.aq_chain:
        if ISiteRoot.providedBy(item):
            return item

```

Checking for the site root

You can check if the current context object is Plone the site root:

```

from Products.CMFCore.interfaces import ISiteRoot

if ISiteRoot.providedBy(context):
    # Special case
else:
    # Subfolder or on a page

```

Navigation root

In Plone, the Plone site root is not necessarily the navigation root (one site can contain many navigation trees for example for the nested subsites).

The navigation root check has the same mechanism as the site root check:

```

from plone.app.layout.navigation.interfaces import INavigationRoot

if INavigationRoot.providedBy(context):
    # Top level, no up navigation
else:
    # Up navigation and breadcrumbs

```

More info

- <https://plone.org/products/plone/roadmap/234>

Getting Zope application server handle

You can also access other sites within the same application server from your code.

Example:

```

app = context.restrictedTraverse('/') # Zope application server root
site = app["plone"] # your plone instance
site2 = app["mysiteid"] # another site

```

Acquisition effect

Sometimes traversal can give you attributes which actually do not exist on the object, but are inherited from the parent objects in the persistent object graph. See *acquisition*.

Default content item

Default content item or view sets some challenges for the traversing, as the object published path and internal path differ.

Below is an example to get the folder of the published object (parent folder for the default item) in page templates:

```
<div tal:define="folder context/@@plone_context_state/canonical_object"
    tal:condition="python:hasattr(folder, 'carousel') and
        hasattr(folder['carousel'],
            'carouselText')">xxx</div>
```

Checking if an item is the site front page

Example:

```
from zope.component import getMultiAdapter

def is_front_page(self):
    """Check if we are in the context of a front page."""
    context_helper = getMultiAdapter((self.context, self.request), name='plone_
↪context_state')
    return context_helper.is_portal_root()
```

Custom traversal

There exist many ways to make your objects traversable:

- `__getitem__()` which makes your objects act like Python dictionary. This is the simplest method and recommended.
- `IPublishTraverse` interface. There is an example below and works for making nice urls and path munging.
- `ITraversable` interface. You can create your own traversing hooks. `zope.traversing.interfaces.ITraversable` provides an interface traversable objects must provide. You need to register `ITraversable` as adapter for your content types. This is only for publishing methods for HTTP requests.
- `__bobo_traverse__()` which is an archaic method from the early 2000s.

Warning: Zope traversal is a minefield. There are different traversers. One is the *ZPublisher traverser* which does HTTP request looks. One is `OFS.Traversable.unrestrictedTraverse()` which is used when you call `traverse` from Python code. Then another case is `zope.tales.expression.PathExpr` which uses a really simple traverser.

Warning: If an `AttributeError` is risen inside a `traverse()` function bad things happen, as Zope publisher specially handles this and raises a `NotFound` exception which will mask the actual problem.

Example using `__getitem__()`:

```
class Viewlets(BrowserView):
    """Expose arbitrary viewlets to traversing by name.
```



```

Exposes viewlets to templates by names.
Example how to render plone.logo viewlet in arbitrary template
code point::

    <div tal:content="context/@@viewlets/plone.logo" />

"""

...

def __getitem__(self, name):
    """
    Allow traversing into viewlets by viewlet name.

    @return: Viewlet HTML output

    @raise: ViewletNotFoundException if viewlet is not found
    """
    viewlet = self.setupViewletByName(name)
    if viewlet is None:
        active_layers = [ str(x) for x in self.request.__provides__.__iro__ ]
        active_layers = tuple(active_layers)
        raise ViewletNotFoundException("Viewlet does not exist by"
            "name %s for the active theme layer set %s."
            "Probably theme interface not registered in "
            "plone.browserlayers. Try reinstalling the theme."
            % (name, str(active_layers)))

    viewlet.update()
    return viewlet.render()

```

Example using IPublishTraverse:

```

from Products.Five.browser import BrowserView
from zope.publisher.interfaces import IPublishTraverse
from zope.interface import implementer
from zope.component import getMultiAdapter
from AccessControl import getSecurityManager
from AccessControl import Unauthorized
from plone import api

@implementer(IPublishTraverse)
class MyUser(BrowserView):
    """Registered as a browser view at '/user', collect the username and
    view name from the url, check security, and display that page. For
    example, '/user/jjohns/log' will look up the log view for user
    'jjohns'
    """
    path = []

    def publishTraverse(self, request, name):
        # stop traversing, we have arrived
        request['TraversalRequestNameStack'] = []
        # return self so the publisher calls this view
        return self

    def __init__(self, context, request):

```

```
"""Once we get to __call__, the path is lost so we
capture it here on initialization
"""

super(MyUser, self).__init__(context, request)
self.section = 'profile-latest' # default page
if len(request.path) == 2:
    [self.section, profileid] = request.path
elif len(self.request.path) == 1:
    self.section = request.path[0]

def __call__(self):
    # do the permission check here, now that Zope has set
    # up the security context. It can't be checked in __init__
    # because the security manager isn't set up on traverse
    self.checkPermission()

    # XXX: still need to check the permission of the view
    try:
        view = api.content.get_view(self.section,
                                    self.context,
                                    self.request)

    except api.exc.InvalidParameterError:
        # just return the default view
        view = api.content.get_view('profile-latest',
                                    self.context,
                                    self.request)

    return view()

def checkPermission(self):
    """You might want to do other stuff"""
    raise Unauthorized
```

More information:

- <http://play.pixelblaster.ro/blog/archive/2006/10/21/custom-traversing-with-five-and-itaversable>

Traverse events

Use `zope.traversing.interfaces.IBeforeTraverseEvent` for register a traversing hook for Plone site object or such.

Example:

```
from Products.CMFCore.interfaces import ISiteRoot
from zope.traversing.interfaces import IBeforeTraverseEvent
from five import grok

@grok.subscribe(ISiteRoot, IBeforeTraverseEvent)
def check_redirect(site, event):
    """
    """
    request = event.request

    # XXX: To something
```

Use `ZPublisher.BeforeTraverse` to register traverse hooks for any objects.

Advanced traversing with search conditions

All Plone content should exist in the *portal_catalog*. Catalog provides fast query access with various indexes to the Plone content.

Other resources

See object [publishing](#).

Publishing

To *publish* an object means to make it available in the Zope traversal graph and URLs.

A published object may have a reverse-mapping of object to path via `getPhysicalPath()` and `absolute_url()` but this is not always the requirement.

You can publish objects by providing a `browser:page` view which implements the `zope.publisher.interfaces.IPublishTraverse` interface.

Example publishers

- A widget to make specified files downloadable: [plone.formwidgets.namedfile.widget](#).

XML-RPC

Description

Using XML-RPC remote call protocol to manipulate Plone site.

Introduction

Zope provides transparent XML-RPC support for any traversable object.

Example:

```
# URL to the object
target = 'http://localhost:8080/plone'

# Call remote method
path = xmlrpclib.ServerProxy(target).getPhysicalPath()
```

Warning: Zope object handles are not transferable across function call boundaries. Thus, you can only call functions with primitive arguments. If you need to call function with object arguments you need to create server side helper code first.

For more information see

- [transmogrifier.ploneremote](#)

Authentication

The simplest way to authenticate the user for XML-RPC calls is to embed HTTP Basic Auth data to URL:

```
# URL to the object
target = 'http://admin:admin@localhost:8080/plone'

# Call remote method
path = xmlrpclib.ServerProxy(target).getPhysicalPath()
```

ZPublisher client

XML-RPC does not marshal objects reliable between remote calls. Getting the real remote object can be done with ZPublisher.Client.Object.

Note: This approach works only for Python clients and needs Zope libraries available at the client side.

Warning: Zope object handles are not transferable across function call boundaries. Thus, you can only call functions with primitive arguments. If you need to call function with object arguments you need to create server side helper code first.

- http://svn.zope.org/Zope/tags/ajung-final-zpt-integration-before-merge-savepoint/utilities/load_site.py?rev=67269&view=auto
- <http://maurits.vanrees.org/weblog/archive/2009/10/lighting-talks-friday#id2>

Web Services API for Plone (wsapi4plone)

This is an add-on product exposes more methods available through Zope's XML-RPC api.

- <https://plone.org/products/wsapi4plone.core>

Importing an Image Using WSAPI

In the following example we retrieve, from the 'Pictures' folder, an image called 'red-wine-glass.jpg', post it to a folder called 'ministries' and give it the name 'theimage'.

```
import os
from xmlrpclib import ServerProxy
from xmlrpclib import Binary

client = ServerProxy("http://username:password@localhost:8080/path/to/plone")

data = open(os.path.join('Pictures', 'red-wine-glass.jpg')).read()

myimage = {'ministries/theimage': [{'title': 'a beautiful wine glass', 'image':
    ↪ Binary(data)}, 'Image']}

output = client.get_object(client.post_object(myimage))
```

For more information see [wsapi4plone.core](#) add-on product adds XML-RPC operations support for Plone.

More information

- <http://www.zope.org/Members/Amos/XML-RPC>

WebDAV

Description

WebDAV is a protocol to manage your site directly from MS Windows Explorer, Mac OS, Linux and so on. Plone supports WebDAV without add-ons, and Plone responds to WebDAV requests out of the box.

Introduction

WebDAV is enabled by default in Plone. A Plone server listening on port 8080 will also accept WebDAV traffic on that port.

Note: WebDAV historically was used mainly for uploading files in bulk to Plone. In Plone 5, this functionality comes standard. For earlier versions, the add-on [collective.wildcardfoldercontents](#) provides this.

Connecting to Plone via WebDAV

For common cases, client-side tools should work reasonably well.

“OS X Mavericks: Connect to a WebDAV server”:

Permissions

The “WebDAV access” permission is required for any user to be able to connect to WebDAV.

To allow Plone users (ie. users created within a Plone site, as opposed to users created in Zope) to connect using WebDAV, go to the Security tab of the Zope (e.g. http://yoursite:8080/manage_access), find the permission “WebDAV access”, check the box for it under the Anonymous column, and press the Save Changes button. This generally grants WebDAV connection access. Normal Plone permissions will take care of who can view or change actual content.

Enabling WebDAV on an extra port in Zope

You can have Plone listen for WebDAV requests on additional ports by modifying your buildout configuration’s client setup to add a WebDAV address:

Here is a short `buildout.cfg` example:

```
[instance]
...
recipe = plone.recipe.zope2instance
...
```

```
webdav-address=1980
...
```

Here is an alternative `buildout.cfg` configuration snippet which might be needed for some WebDAV clients:

```
[instance]
...
zope-conf-additional =
    enable-ms-author-via on
    <webdav-source-server>
    address YOUR_SERVER_PUBLIC_IP_ADDRESS_HERE:1980
    force-connection-close off
    </webdav-source-server>
```

These snippets will be in the **generated** `parts/instance/etc/zope.conf` after buildout has been re-run.

This will enable the WebDAV server on <http://www.mydomain.com:1980/>.

Note: You cannot use this URL in your web browser, just in WebDAV clients.

Using the web browser will give you an error message `AttributeError: manage_FTPget`. You could also just run the WebDAV server on `localhost` with address 1980, forcing you to either use a WebDAV client locally or proxy WebDAV through Apache.

Disabling WebDAV

You can't disable WebDAV in Plone itself; it's tightly integrated in Zope. You could take away the "WebDAV access" permission from everyone, but the Zope server will still answer each request.

What you can do is make your web server filter out the WebDAV commands; this will stop WebDAV requests before they reach your Zope server.

Nginx

For nginx, this is done by adding:

```
dav_methods off
```

to the server block in your `nginx.conf`. (<http://wiki.nginx.org/HttpDavModule>)

If you do not use the `HttpDavModule`, you can add:

```
limit_except GET POST {
    deny    all;
}
```

to the location block.

Apache

For Apache, you can use the `limit` statement, see <http://httpd.apache.org/docs/current/mod/core.html#limit>

See also:

“How can I stop people accessing a Plone server via WebDAV?”

Supporting WebDAV in your custom content

Please read more about it in the [Dexterity WebDAV manual](#).

FTP

Plone/Zope supports FTP in the default configuration.

FTP support is not very well maintained. WebDAV protocol is recommended over FTP.

Enabling FTP

See [zope2instance recipe](#).

Views, viewlets and layers

View and viewlet patterns used to create dynamic pages in plone.

Views

Description

Rendering HTML pages in Plone using the Zope 3 *view* pattern.

Introduction

Plone/Zope uses a *view* pattern to output dynamically generated HTML pages.

Views are the basic elements of modern Python web frameworks. A view runs code to setup Python variables for a rendering template. Output is not limited to HTML pages and snippets, but may contain *JSON*, file download payloads, or other data formats.

Views are usually a combination of:

- a Python class, which performs the user interface logic setup, and a
- corresponding *ZPT* page template, or direct Python string output.

By keeping as much of the view logic in a separate Python class as we can and making the page template as simple as possible, better component readability and reuse is achieved. You can override the Python logic or the template file, or both.

When you are working with Plone, the most usual view type is `BrowserView` from the [Products.Five](#) package, but there are others.

Each `BrowserView` class is a Python callable. The `BrowserView.__call__()` method acts as an entry point to executing the view code. From Zope’s point of view, even a function would be sufficient, as it is a callable.

More information

- [Mastering Plone Training](#) has several chapters on views.

View components

Views are Zope Component Architecture (*ZCA*) *multi-adapter registrations*.

Views are looked up by name. The Zope publisher always does a view lookup, instead of traversing, if the name to be traversed is prefixed with @@.

Views are resolved with three inputs:

context Any class/interface for which the view applies. If not given, `zope.interface.Interface` is used (corresponds to a registration `for="*"`). Usually this is a content item instance.

request The current HTTP request. Interface `zope.publisher.interfaces.browser.IBrowserRequest` is used.

layer Theme layer and addon layer interface. If not given, `zope.publisher.interfaces.browser.IDefaultBrowserLayer` is used.

Views return HTTP request payload as the output. Returned strings are turned to HTML page responses.

Views can be any Python class taking in (context, request) construction parameters. Minimal view would be:

```
class MyView(object):

    def __init__(self, context, request):
        self.context = context
        self.request = request

    def __call__(self):
        return "Hello world. You are rendering this view at the context of %s" % \
            self.context
```

However, in the most of cases

- Full Plone page views are subclass of `Products.Five.browser.BrowserView` which is a wrapper class. It wraps `zope.publisher.browser.BrowserView` and adds an acquisition (parent traversal) support for it.
- Views have `index` attribute which points to *TAL page template* responsible rendering the HTML code. You get the HTML output by doing `self.index()` and page template gets a context argument `view` pointing to the view class instance. `index` value is usually instance of `Products.Five.browser.pagetemplate.ViewPageTemplateFile` (full Plone pages) or `zope.pagetemplate.pagetemplatefile.PageTemplateFile` (HTML snippets, no acquisition)
- View classes should implement *interface* `zope.browser.interfaces.IBrowserView`

Views rendering page snippets and parts can be subclasses of `zope.publisher.browser.BrowserView` directly as snippets might not need acquisition support which adds some overhead to the rendering process.

Customizing views

To customize existing Plone core or add-on views you have different options.

- Usually you can simply override the related page template file (`.pt`).
- Sometimes you need to change the related Python view class code also. In this case, you override the Python class by using your own add-on which installs a view class replacement using add-on layer.

Overriding view template

Follow instructions how to [use z3c.jbot](#) to override templates.

Overriding view class

Here is a short introduction on finding how existing views are defined.

First, you go to `portal_types` to see what views have been registered for a particular content type.

For example, if you want to override the *Tabular* view of a *Folder*, you find out that it is registered as the handler for `/folder_tabular_view`.

You look for both `folder_tabular_view` old-style page templates and `@@folder_tabular_view` `BrowserView` ZCML registrations in the Plone source tree — it can be either.

Example of how to search for this using UNIX tools (assuming that [collective.recipe.omelette](#) is in use, to keep included code together):

```
# find old style .pt files:
find parts/omelette -follow -name "folder_tabular_view*"
# find new style view registrations in ZCML files:
grep -ri --include="*.zcml" folder_tabular_view parts/omelette
```

The `folder_tabular_view` is found in the *skin layer* called `plone_content` in the CMFPlone product.

More info:

- [How to override old style page templates](#)

Creating and registering a view

This shows how to create and register view in a Zope 3 manner.

Creating a view

Create your add-on product using [Dexterity project template](#).

Python logic code

Add the file `yourcompany.app/yourcompany/app/browser/views.py`:

```
""" Example view
"""

# Zope imports
from zope.interface import Interface
from Products.Five.browser import BrowserView
from Products.Five.browser.pagetemplatefile import ViewPageTemplateFile

class MyView(BrowserView):
    """ Render the title and description of item only (example)
    """
    index = ViewPageTemplateFile("myview.pt")
```

Warning: Do not attempt to run any code in the `__init__()` method of a view. If this code fails and an exception is raised, the `zope.component` machinery remaps this to a “View not found” exception or traversal error.

Additionally, view class may be instantiated in other places than where you intended to render the view. For example, `plone.app.contentmenu` does this when creating the menu to select a view layout. This will result in the `__init__()` being called on unexpected contexts, probably wasting a lot of time.

Instead, use a pattern where you have a `setup()` or similar method which `__call__()` or view users can explicitly call.

Registering a view

Zope 3 views are registered in *ZCML*, an XML-based configuration language. Usually, the configuration file, where the registration done, is called `yourapp.package/yourapp/package/browser/configure.zcml`.

The following example registers a new view (see below for comments):

```
<configure
  xmlns="http://namespaces.zope.org/zope"
  xmlns:browser="http://namespaces.zope.org/browser"
  >

  <browser:page
    for="*"
    name="test"
    permission="zope2.Public"
    class=".views.MyView"
  />

</configure>
```

for specifies which content types receive this view. `for="*"` means that this view can be used for any content type. This is the same as registering views to the `zope.interface.Interface` base class.

name is the name by which the view is exposed to traversal and `getMultiAdapter()` look-ups. If your view’s name is `test`, then you can render it in the browser by calling <http://yourhost/site/page/@@test>

permission is the permission needed to access the view. When an HTTP request comes in, the currently logged in user’s access rights in the current context are checked against this permission. See *Security chapter* for Plone’s out-of-the-box permissions. Usually you want have `zope2.View`, `cmf.ModifyPortalContent`, `cmf.ManagePortal` or `zope2.Public` here.

class is a Python dotted name for a class based on `BrowserView`, which is responsible for managing the view. The Class’s `__call__()` method is the entry point for view processing and rendering.

Note: You need to declare the browser namespace in your `configure.zcml` to use browser configuration directives.

The view in question is not registered against any *layer*, it is immediately available after restart without need to run *Add/remove in Site setup*.

Page template

Create a *page template for your view*. Create a file `myview.pt` file in `yourcompany.app/yourcompany/app/browser/templates` and add the template:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:metal="http://xml.zope.org/namespaces/metal"
      xmlns:tal="http://xml.zope.org/namespaces/tal"
      xmlns:i18n="http://xml.zope.org/namespaces/i18n"
      metal:use-macro="context/main_template/macros/master">

  <metal:block fill-slot="content-core">
    XXX - this text comes below title and description
  </metal:block>

</html>
```

When you restart to Plone (or use *auto-restart add-on*) the view should be available through your browser.

Accessing your newly created view

Now you can access your view within the news folder:

```
http://localhost:8080/Plone/news/myview
```

... or on a site root:

```
http://localhost:8080/Plone/myview
```

... or on any other content item.

You can also use the `@@` notation at the front of the view name to make sure that you are looking up a *view*, and not a content item that happens to have the same id as a view:

```
http://localhost:8080/Plone/news/@@myview
```

More info

- <https://plone.org/products/dexterity/documentation/manual/five.grok/browser-components/views>

Content slots

Available *slot* options you can use for `<metal fill-slot="">` in your template which inherits from `<html metal:use-macro="context/main_template/macros/master">`:

content render edit border yourself

main overrides main slot in main template; you must render title and description yourself

content-title title and description prerendered, Plone version > 4.x

content-core content body specific to your view, Plone version > 4.x

header A slot for inserting content above the title; may be useful in conjunction with `content-core` slot if you wish to use the stock `content-title` provided by the main template.

Relationship between views and templates

The ZCML `<browser:view template="">` directive will set the `index` class attribute.

The default view's `__call__()` method will return the value returned by a call to `self.index()`.

Example: this ZCML configuration:

```
<browser:page
    for="*"
    name="test"
    permission="zope2.Public"
    class=".views.MyView"
/>
```

and this Python code:

```
from Products.Five.browser import BrowserView
from Products.Five.browser.pagetemplatefile import ViewPageTemplateFile

class MyView(BrowserView):

    index = ViewPageTemplateFile("my-template.pt")
```

is equal to this ZCML configuration:

```
<browser:page
    for="*"
    name="test"
    permission="zope2.Public"
    class=".views.MyView"
    template="my-template.pt"
/>
```

and this Python code:

```
class MyView(BrowserView):
    pass
```

Rendering of the view is done as follows:

```
from Products.Five.browser.pagetemplatefile import ViewPageTemplateFile

class MyView(BrowserView):

    # This may be overridden in ZCML
    index = ViewPageTemplateFile("my-template.pt")

    def render(self):
        return self.index()

    def __call__(self):
        return self.render()
```

Overriding a view template at run-time

Below is a sample code snippet which allows you to override an already constructed `ViewPageTemplateFile` with a chosen file at run-time:

```
import plone.z3cform
from zope.app.pagetemplate import ViewPageTemplateFile as Zope3PageTemplateFile
from zope.app.pagetemplate.viewpagetemplatefile import BoundPageTemplate

# Construct template from a file which lies in a certain package
template = Zope3PageTemplateFile(
    'subform.pt',
    os.path.join(
        os.path.dirname(plone.z3cform.__file__),
        "templates"))

# Bind template to context:
# make the template callable with template() syntax and context
form_instance.template = BoundPageTemplate(template, form_instance)
```

Several templates per view

You can bind several templates to one view and render them individually. This is useful for reusable templating, or when you subclass your functional views.

Example using `five.grok`:

```
class CourseTimetables(grok.View):

    # For communicating state variables from Python code to JavaScript
    jsHeaderTemplate = grok.PageTemplateFile("templates/course-timetables-fees-js-
    ↳ snippet.pt")

    def renderJavascript(self):
        return self.jsHeaderTemplate.render(self)
```

And then call in the template:

```
<metal:javascriptslot fill-slot="javascript_head_slot">
    <script tal:replace="structure view/renderJavascript" />
</metal:javascriptslot>
```

View `__init__()` method special cases

The Python constructor method of the view, `__init__()`, is special. You should never try to put your code there. Instead, use helper method or lazy construction design pattern if you need to set-up view variables.

The `__init__()` method of the view might not have an *acquisition chain* available, meaning that it does not know the parent or hierarchy where the view is.

This information is set after the constructor have been run. All Plone code which relies on acquisition chain, which means almost all Plone helper code, does not work in `__init__()`. Thus, the called Plone API methods return `None` or tend to throw exceptions.

Layers

Views can be registered against a specific *layer* interface. This means that views are only looked up if the specified layer is in use. Since one Zope application server can contain multiple Plone sites, layers are used to determine which Python code is in effect for a given Plone site.

A layer is in use when:

- a theme which defines that layer is active, or
- if a specific add-on product which defines that layer is installed.

You should register your views against a certain layer in your own code.

For more information, see

- *browser layers*

Register and unregister view directly using zope.component architecture

Example how to register:

```
import zope.component
import zope.publisher.interfaces.browser

zope.component.provideAdapter(
    # Our class
    factory=TestingRedirectHandler,
    # (context, request) layers for multiadapter lookup
    # We provide None as layers are not used
    adapts=(None, None),
    # All views are registered as IBrowserView interface
    provides=zope.publisher.interfaces.browser.IBrowserView,
    # View name
    name='redirect_handler')
```

Example how to unregister:

```
# Dynamically unregister a view
gsm = zope.component.getGlobalSiteManager()
gsm.unregisterAdapter(factory=TestingRedirectHandler,
                      required=(None, None),
                      provided=zope.publisher.interfaces.browser.IBrowserView,
                      name="redirect_handler")
```

Content type, mimetype and Template start tag

If you need to produce non-(X)HTML output, here are some resources:

- <http://plone.293351.n2.nabble.com/Setting-a-mime-type-on-a-Zope-3-browser-view-td4442770.html>

Zope ViewPageTemplateFile vs. Five ViewPageTemplateFile

Warning: There are two different classes that share the same `ViewPageTemplateFile` name.

- Zope `BrowserView` source code.
- **Five version.** `Products.Five` is a way to access some Zope 3 technologies from the Zope 2 codebase, which is used by Plone.

Difference in code:

```
from Products.Five.browser.pagetemplatefile import ViewPageTemplateFile
```

vs.:

```
from zope.app.pagetemplate import ViewPageTemplateFile
```

The difference is that the *Five* version supports:

- Acquisition.
- The `provider:` TAL expression.
- Other Plone-specific TAL expression functions like `test()`.
- Usually, Plone code needs the Five version of `ViewPageTemplateFile`.
- Some subsystems, notably the `z3c.form` package, expect the Zope 3 version of `ViewPageTemplateFile` instances.

Overriding a view class in a product

Most of the code in this section is copied from a [tutorial by Martin Aspeli \(on slideshare.net\)](#). The main change is that, at least for Plone 4, the interface should subclass `plone.theme.interfaces.IDefaultPloneLayer` instead of `zope.interface.Interface`.

In this example we override the `@@register` form from the `plone.app.users` package, creating a custom form which subclasses the original.

- Create an interface in `interfaces.py`:

```
from plone.theme.interfaces import IDefaultPloneLayer

class IExamplePolicy(IDefaultPloneLayer):
    """ A marker interface for the theme layer
    """
```

- Then create `profiles/default/browserlayer.xml`:

```
<layers>
  <layer
    name="example.policy.layer"
    interface="example.policy.interfaces.IExamplePolicy"
  />
</layers>
```

- Create `browser/configure.zcml`:

```
<configure
  xmlns="http://namespaces.zope.org/zope"
  xmlns:browser="http://namespaces.zope.org/browser"
  i18n_domain="example.policy">
  <browser:page
    name="register"
    class=".customregistration.CustomRegistrationForm"
    permission="cmf.AddPortalMember"
    for="plone.app.layout.navigation.interfaces.INavigationRoot"
    layer="example.policy.interfaces.IExamplePolicy"
  />
</configure>
```

Note: We've retained the permissions and marker interface of the original view. You may provide a specific permission or marker interface instead of these as your product requires.

- Create `browser/customregistration.py`:

```
from plone.app.users.browser.register import RegistrationForm

class CustomRegistrationForm(RegistrationForm):
    """ Subclass the standard registration form
    """
```

Helper views

Not all views need to return HTML output, or output at all. Views can be used as helpers in the code to provide APIs to objects. Since views can be overridden using layers, a view is a natural plug-in point which an add-on product can customize or override in a conflict-free manner.

View methods are exposed to page templates and such, so you can also call view methods directly from a page template, not only from Python code.

More information

- *Context helpers*
- *Expressions*

Historical perspective

Often, the point of using helper views is that you can have reusable functionality which can be plugged in as one-line code around the system. Helper views also get around the following limitations:

- TAL security.
- Limiting Python expression to one line.
- Not being able to import Python modules.

Note: Using `RestrictedPython` scripts (creating Python through the Management Interface) and Zope 2 Extension modules is discouraged. The same functionality can be achieved with helper views, with less potential pitfalls.

Reusing view template snippets or embedding another view

To use the same template code several times you can either:

- create a separate `BrowserView` for it and then call this view (see *Accessing a view instance in code* below);
- share a `ViewPageTemplateFile` instance between views and using it several times.

Note: The Plone 2.x way of doing this with TAL template language macros is discouraged as a way to provide reusable functionality in your add-on product. This is because macros are hardwired to the TAL template language, and referring to them outside templates is difficult.

If you ever need to change the template language, or mix in other template languages, you can do better when templates are a feature of a pure Python based view, and not vice versa.

Here is an example of how to have a view snippet which can be used by subclasses of a base view class. Subclasses can refer to this template at any point of the view rendering, making it possible for subclasses to have fine-tuned control over how the template snippet is represented.

Related Python code:

```
from Products.Five import BrowserView
from Products.Five.browser.pagetemplatefile import ViewPageTemplateFile

class ProductCardView(BrowserView):
    """
    End user visible product card presentation.
    """
    implements(IProductCardView)

    # Nested template which renders address box + buy button
    summary_template = ViewPageTemplateFile("summarybox.pt")

    def renderSummary(self):
        """ Render summary box

        @return: Resulting HTML code as Python string
        """
        return self.summary_template()
```

Then you can render the summary template in the main template associated with `ProductCardView` by calling the `renderSummary()` method and TAL non-escaping HTML embedding.

```
<h1 tal:content="context/Title" />

<div tal:replace="structure view/renderSummary" />

<div class="description">
```

```
<div tal:condition="python:context.Description().decode('utf-8') != u'None'"
↳tal:replace="structure context/Description" />
</div>
```

The `summarybox.pt` itself is a piece of HTML code without the Plone decoration frame (`main_template/master` etc. macros). Make sure that you declare the `il8n:domain` again, or the strings in this template will not be translated.

```
<div class="summary-box" il8n:domain="your.package">
...
</div>
```

Accessing a view instance in code

You need to get access to the view in your code if you are:

- calling a view from inside another view, or
- calling a view from your unit test code.

Below are two different approaches for that.

By using `getMultiAdapter()`

This is the most efficient way in Python.

Example:

```
from Acquisition import aq_inner
from zope.component import getMultiAdapter

def getView(context, request, name):
    # Remove the acquisition wrapper (prevent false context assumptions)
    context = aq_inner(context)
    # May raise ComponentLookupError
    view = getMultiAdapter((context, request), name=name)
    # Add the view to the acquisition chain
    view = view.__of__(context)
    return view
```

By using traversal

Traversal is slower than directly calling `getMultiAdapter()`. However, traversal is readily available in templates and `RestrictedPython` modules.

Example:

```
def getView(context, name):
    """ Return a view associated with the context and current HTTP request.

    @param context: Any Plone content object.
    @param name: Attribute name holding the view name.
    """
```

```

try:
    view = context.unrestrictedTraverse("@@" + name)
except AttributeError:
    raise RuntimeError("Instance %s did not have view %s" % (str(context), name))

view = view.__of__(context)

return view

```

You can also do direct view look-ups and method calls in your template by using the @@-notation in traversing.

```

<div tal:attributes="lang context/@@plone_portal_state/current_language">
    We look up lang attribute by using BrowserView which name is "plone_portal_state"
</div>

```

Use a skin-based template in a Five view

Use `aq_acquire(object, template_name)`.

Example: Get an object by its path and render it using its default template in the current context.

```

from Acquisition import aq_base, aq_acquire
from Products.Five.browser import BrowserView

class TelescopeView(BrowserView):
    """
    Renders an object in a different location of the site when passed the
    path to it in the querystring.
    """
    def __call__(self):
        path = self.request["path"]
        target_obj = self.context.restrictedTraverse(path)
        # Strip the target_obj of context with aq_base.
        # Put the target in the context of self.context.
        # getDefaultLayout returns the name of the default
        # view method from the factory type information
        return aq_acquire(aq_base(target_obj).__of__(self.context),
                        target_obj.getDefaultLayout())()

```

Listing available views

This is useful for debugging purposes:

```

from plone.app.customerize import registration
from zope.publisher.interfaces.browser import IBrowserRequest

# views is generator of zope.component.registry.AdapterRegistration objects
views = registration.getViews(IBrowserRequest)

```

Listing all views of certain type

How to filter out views which provide a certain interface:

```
from plone.app.customerize import registration
from zope.publisher.interfaces.browser import IBrowserRequest

# views is generator of zope.component.registry.AdapterRegistration objects
views = registration.getViews(IBrowserRequest)

# Filter out all classes which implement a certain interface
views = [ view.factory for view in views if IBlocksView.implementedBy(view.factory) ]
```

Default view of a content item

Objects have views for default, view, edit, and so on.

The distinction between the *default* and *view* views are that for files, the default can be *download*.

The default view ...

- This view is configured in *portal_types*.
- This view is rendered when a content item is called — even though they are objects, they have the `__call__()` Python method defined.

If you need to get a content item's view for page rendering explicitly, you can do it as follows:

```
def viewURLFor(item):
    cstate = getMultiAdapter((item, item.REQUEST),
                             name='plone_context_state')
    return cstate.view_url()
```

More info:

- *Context helpers and utilities*
- <http://plone.293351.n2.nabble.com/URL-to-content-view-tp6028204p6028204.html>

Allowing the contentmenu on non-default views

In general, the contentmenu (where the actions, display views, factory types, workflow, and other dropdowns are) is not shown on non-default views. There are some exceptions, though.

If you want to display the contentmenu in such non-default views, you have to mark them with the `IViewView` interface from `plone.app.layout` either by letting the class provide `IViewView` by declaring it with `zope.component.implements` or by configuring it via ZCML like so:

```
<class class="dotted.path.to.browser.view.class">
  <implements interface="plone.app.layout.interfaces.IViewView" />
</class>
```

Views and automatic member variable acquisition wrapping

View class instances will automatically assign themselves as a parent for all member variables. This is because five package based views inherit from `Acquisition.Implicit` base class.

E.g. you have a `Basket` content item with `absolute_url()` of:

```
http://localhost:9666/isleofback/sisalto/matkasuunnitelmat/
↳ d59ca034c50995d6a77cacbe03e718de
```

Then if you use this object in a view code's member variable assignment in e.g. `Viewlet.update()` method:

```
self.basket = my_basket
```

... this will mess up the Basket content item's acquisition chain:

```
<Basket at /isleofback/sisalto/yritykset/katajamaan_taksi/
↳ d59ca034c50995d6a77cacbe03e718de>
```

This concerns views, viewlets and portlet renderers. It will, for example, make the following code to fail:

```
self.obj = self.context.reference_catalog.lookupObject(value)
return self.obj.absolute_url() # Acquisition chain messed up, getPhysicalPath() fails
```

One workaround to avoid this mess is to use `aq_inner` when accessing `self.obj` values:

- <http://stackoverflow.com/a/11755348/315168>

Viewlets

Description

Viewlets are parts of the page in Plone page rendering process. You can create, hide and shuffle them freely.

Introduction

Viewlets are view snippets which will render a part of the HTML page. Viewlets provide conflict-free way to contribute new user-interface actions and HTML snippets to Plone pages.

Each viewlet is associated with a viewlet manager. To add viewlets to your HTML code you first need to add them to a viewlet manager, which allows you to shuffle viewlets around through-the-web.

What viewlets do

- Viewlets are managed using `/@@manage-viewlets` page
- Viewlets can be shown and hidden through-the-web
- Viewlets can be reordered (limited to reordering within container in Plone 3.x)
- Viewlets can be registered and overridden in a theme specific manner *using layers*
- Viewlets have `update()` and `render()` methods
- Viewlets should honour `zope.contentprovider.interfaces.IContentProvider` call contract.

A viewlet can be configured so that it is only available for:

- a certain interface, typically a content type (`for=` in ZCML)
- a certain view (`view=` in ZCML)

More info

- Plone 4 Viewlet and viewlet manager reference
- ZCML viewlet definition.
- <https://pypi.python.org/pypi/zope.viewlet/>

Finding viewlets

There are two through-the-web tools to start looking what viewlets are available on your installation. The available viewlets may depend on installed Plone version and installed add-ons.

- The `portal_view_customizations` tool in the Management Interface will show you viewlet registrations (and the viewlet managers they are registered for). As with views, you can hover over the viewlet name to see where it is registered in a tool tip.
- To discover the name of a particular viewlet, you can use the `@manage-viewlets` view, e.g. <http://localhost:8080/plone/@manage-viewlets>.

Creating a viewlet

A viewlet consists of

- Python class
- Page template (.pt) file
- A *browser layer* defining which add-on product must be installed, so that the viewlet is rendered
- A ZCML directive to register the viewlet to a correct viewlet manager with a correct layer

Re-using code from a View

In the case where you might want a Viewlet and View to share the same code, remember that the View instance is available in the Viewlet under the `view` attribute.

Thus, you can use `self.view` to get the view, and then use its methods.

Stock viewlets

These can be found in `plone.app.layout.viewlet` module.

The language selector lives in `plone.app.i18n.locales.browser`, but it is a *view*. Don't know why.

Creating a viewlet manager

Viewlet managers contain viewlets. A viewlet manager is itself a Zope 3 interface which contains an `OrderedViewletManager` implementation. `OrderedViewletManagers` store the order of the viewlets in the site database and provide the fancy `/@@manage-viewlets` output.

A viewlet manager can be rendered in a page template code using the following expression:

```
<div tal:replace="structure provider:viewletmanagerid" />
```

Each viewlet manager allows you to shuffle viewlets inside a viewlet manager. This is done by using `/@@manage-viewlets` view. These settings are stored in the site database, so a good practice is to export `viewlets.xml` using `portal_setup` and then include the necessary bits of this `viewlets.xml` with your add-on installer so that when your add-on is installed, the viewlet configuration is changed accordingly.

Note: You cannot move viewlets between viewlet managers. I know it sucks, but life is hard and Plone is harder. Hide viewlets in one manager using `/@@manage-viewlets` and `viewlets.xml` export, then re-register the same viewlet to a new manager.

Viewlet managers are based on `zope.viewlet.manager.ViewletManager` and `plone.app.viewletmanager.manager.OrderedViewletManager`.

More info

- <https://github.com/zopefoundation/zope.viewlet/blob/3.7.2/src/zope/viewlet/viewlet.py>
- http://docs.plone.org/old-reference-manuals/plone_3_theming/elements/viewletmanager/anatomy.html

Creating a viewlet manager

Usually viewlet managers are dummy interfaces and the actual implementation comes from `plone.app.viewletmanager.manager.OrderedViewletManager`.

In this example we put two viewlets in a new viewlet manager so that we can properly CSS float then and close this float.

Note: This example uses extensive Python module nesting: `plonetheme.yourtheme.browser.viewlets` is just too deep. You really don't need to do some many levels, but the original `plone3_theme` pasteur templates do it in bad way. One of Python golden rules is that flat is better than nested. You can just dump everything to the root of your `plonetheme.yourtheme` package.

In your `browser/viewlets/manager.py` or similar file add:

```
<browser:viewletManager
    name="plonetheme.yourtheme.headerbottommanager"
    provides="plonetheme.yourtheme.browser.viewlets.manager.
↪ IHeaderBottomViewletManager"
    class="plone.app.viewletmanager.manager.OrderedViewletManager"
    layer="plonetheme.yourtheme.browser.interfaces.IThemeSpecific"
    permission="zope2.View"
    template="headerbottomviewletmanager.pt"
/>
```

Then in `browser/viewlets/configure.zcml`:

```
<browser:viewletManager
    name="plonetheme.yourock.browser.viewlets.MyViewletManager"
    provides=".viewlets.MyViewletManager"
    class="plone.app.viewletmanager.manager.OrderedViewletManager"
    layer="plonetheme.yourock.interfaces.IThemeLayer"
    permission="zope2.View"
/>
```

Optionally you can include a template which renders some wrapping HTML around viewlets. `browser/viewlets/headerbottomviewletmanager.pt`:

```
<div id="header-bottom">
  <tal:comment replace="nothing">
    <!-- Rendeder all viewlets inside this manager.
    Pull viewlets out of the manager and render then one-by-one
    -->
  </tal:comment>

  <tal:viewlets repeat="viewlet view/viewlets">
    <tal:viewlet replace="structure python:viewlet.render()" />
  </tal:viewlets>

  <div style="clear:both"><!-- --></div>
</div>
```

And then re-register some stock viewlets against your new viewlet manager in *browser/viewlets/configure.zcml*:

```
<!-- Re-register two stock viewlets to the new manager -->

<browser:viewlet
  name="plone.path_bar"
  for="*"
  manager="plonetheme.yourtheme.browser.viewlets.manager.IHeaderBottomViewletManager
  ↪"
  layer="plonetheme.yourtheme.browser.interfaces.IThemeSpecific"
  class="plone.app.layout.viewlets.common.PathBarViewlet"
  permission="zope2.View"
/>

<!-- This is a customization for rendering the a bit different language selector -->
<browser:viewlet
  name="plone.app.i18n.locales.languageselector"
  for="*"
  manager="plonetheme.yourtheme.browser.viewlets.manager.IHeaderBottomViewletManager
  ↪"
  layer="plonetheme.yourtheme.browser.interfaces.IThemeSpecific"
  class=".selector.LanguageSelector"
  permission="zope2.View"
/>
```

Now, we need to render our viewlet manager somehow. One place to do it is in a *main_template.pt*, but because we need to add this HTML output to a header section which is produced by *another* viewlet manager, we need to create a new viewlet just for rendering our viewlet manager. Yo dawg - we put viewlets in your viewlets so you can render viewlets!

browser/viewlets/headerbottom.pt:

```
<tal:comment replace="nothing">
  <!-- Render our precious viewlet manager -->
</tal:comment>
<tal:render-manager replace="structure provider:plonetheme.yourtheme.
  ↪headerbottommanager" />
```

Only six files needed to change a bit of HTML code - welcome to the land of productivity! On the top of this you also need to create a new *viewlets.xml* export for your theme.

More info

- <https://plone.org/documentation/manual/theme-reference/elements/viewletmanager/override>

Viewlet behavior

Viewlets have two important methods

1. `update()` - set up all variables
2. `render()` - generate the resulting HTML code by evaluating the template with context variables set up in `update()`

These methods should honour `zope.contentprovider.interfaces.IContentProvider` call `contract`.

See

- <https://github.com/zopefoundation/zope.contentprovider/blob/3.7.2/src/zope/contentprovider/interfaces.py>
- <https://github.com/plone/plone.app.layout/blob/master/plone/app/layout/viewlets/common.py>

Creating a viewlet using Python code and ZCML

Here is an example code which extends an existing Plone base viewlet (found from `plone.app.layout.viewlets.base` package) and then puts this viewlet to a one of viewlet managers using *ZCML*.

Example Python code for `viewlets.py`:

```
"""
    Facebook like viewlet for Plone.

    http://mfabrik.com
"""

import urllib

from plone.app.layout.viewlets import common as base

class LikeViewlet(base.ViewletBase):
    """ Add a Like button

    http://developers.facebook.com/docs/reference/plugins/like
    """

    def constructParameters(self):
        """ Create HTTP GET query parameters send to Facebook used to render the_
        ↪button.

        href=http%253A%252F%252Fexample.com%252Fpage%252Fto%252Flike&
        ↪layout=standard&show_faces=true&width=450&action=like&font&
        ↪colorscheme=light&height=80
        """

        context = self.context.aq_inner
        href = context.absolute_url()

        params = {
            "href" : href,
            "layout" : "standard",
            "show_faces" : "true",
```

```

        "width" : "450",
        "height" : "40",
        "action" : "like",
        "colorscheme" : "light",
    }

    return params

def getIFrameSource(self):
    """
    @return: <iframe src=""> string
    """
    params = self.constructParameters()
    return "http://www.facebook.com/plugins/like.php" + "?" + urllib.
    ↪urlencode(params)

def getStyle(self):
    """ Construct CSS style for Like-button IFRAME.

    Use width and height from contstructParameters()

    style="border:none; overflow:hidden; width:450px; height:80px;"

    @return: style="" for <iframe>
    """
    params = self.constructParameters()
    return "margin-left: 10px; border:none; overflow:hidden; width:{}px; height:{}
    ↪px;".format(params["width"], params["height"])

```

Then a sample page template (like.pt). You can use TAL template variable *view* to refer to your viewlet class instance:

```

<iframe scrolling="no"
    frameborder="0"
    allowTransparency="true"
    tal:attributes="src view/getIFrameSource; style view/getStyle"
>
</iframe>

```

Registering a viewlet using ZCML

Example configuration ZCML snippets below. You usually <viewlet> to *browser/configure.zcml* folder.

```

<configure
    xmlns="http://namespaces.zope.org/zope"
    xmlns:five="http://namespaces.zope.org/five"
    xmlns:browser="http://namespaces.zope.org/browser"
    xmlns:genericsetup="http://namespaces.zope.org/genericsetup"
    i18n_domain="mfabrik.like">

    <browser:viewlet
        name="mfabrik.like"
        manager="plone.app.layout.viewlets.interfaces.IBelowContent"
        template="like.pt"
        layer="mfabrik.like.interfaces.IAddOnInstalled"
        permission="zope2.View"
    >

```

```

    />

</configure>

```

Conditionally rendering viewlets

There are two primary methods to render viewlets only on some pages

- Register viewlet against some marker interface or content type class - the viewlet is rendered on this content type only. You can use *dynamic marker interfaces* to toggle interface on some individual pages through the Management Interface.
- Hard-code a condition to your viewlet in Python code.

Below is an example of overriding a render() method to conditionally render your viewlet:

```

import Acquisition
from zope.component import getUtility

from plone.app.layout.viewlets import common as base
from plone.registry.interfaces import IRegistry

class LikeViewlet(base.ViewletBase):
    """ Add a Like button

    http://developers.facebook.com/docs/reference/plugins/like
    """

    def isEnabledOnContent(self):
        """
        @return: True if the current content type supports Like-button
        """
        registry = getUtility(IRegistry)
        content_types = registry['mfabrik.like.content_types']

        # Don't assume that all content items would have portal_type attribute
        # available (might be changed in the future / very specialized content)
        current_content_type = portal_type = getattr(
            Acquisition.aq_base(self.context), 'portal_type', None)

        # Note that plone.registry keeps values as unicode strings
        # make sure that we have one also
        current_content_type = unicode(current_content_type)

        return current_content_type in content_types

    def render(self):
        """ Render viewlet only if it is enabled.

        """

        # Perform some condition check
        if self.isEnabledOnContent():
            # Call parent method which performs the actual rendering
            return super(LikeViewlet, self).render()

```

```
else:
    # No output when the viewlet is disabled
    return ""
```

Rendering viewlet by name

Below is a complex example how to expose viewlets without going through a viewlet manager.

```
from Acquisition import aq_inner
import zope.interface

from plone.app.customerize import registration

from Products.Five.browser import BrowserView

from zope.traversing.interfaces import ITraverser, ITraversable
from zope.publisher.interfaces import IPublishTraverse
from zope.publisher.interfaces.browser import IBrowserRequest
from zope.viewlet.interfaces import IViewlet
from zExceptions import NotFound

class Viewlets(BrowserView):
    """ Expose arbitrary viewlets to traversing by name.

    Exposes viewlets to templates by names.

    Example how to render plone.logo viewlet in arbitrary template code point::

        <div tal:content="context/@@viewlets/plone.logo" />

    """
    zope.interface.implements(ITraversable)

    def getViewletByName(self, name):
        """ Viewlets allow through-the-web customizations.

        Through-the-web customization magic is managed by five.customerize.
        We need to think of this when looking up viewlets.

        @return: Viewlet registration object
        """
        views = registration.getViews(IBrowserRequest)

        for v in views:
            if v.provided == IViewlet:
                # Note that we might have conflicting BrowserView with the same name,
                # thus we need to check for provided
                if v.name == name:
                    return v

        return None

    def setupViewletByName(self, name):
        """ Constructs a viewlet instance by its name.
```

```

Viewlet update() and render() method are not called.

@return: Viewlet instance of None if viewlet with name does not exist
"""
context = aq_inner(self.context)
request = self.request

# Perform viewlet registration look-up
# from adapters registry
reg = self.getViewletByName(name)
if reg == None:
    return None

# factory method is responsible for creating the viewlet instance
factory = reg.factory

# Create viewlet and put it to the acquisition chain
# Viewlet need initialization parameters: context, request, view
try:
    viewlet = factory(context, request, self, None).__of__(context)
except TypeError:
    # Bad constructor call parameters
    raise RuntimeError(
        "Unable to initialize viewlet {}. Factory method {} call failed."
        .format(name, str(factory)))

return viewlet

def traverse(self, name, further_path):
    """
    Allow traversing into viewlets by viewlet name.

    @return: Viewlet HTML output

    @raise: RuntimeError if viewlet is not found
    """

    viewlet = self.setupViewletByName(name)
    if viewlet is None:
        raise NotFound("Viewlet does not exist by name {} for theme layer".
            ↪format(name))

    viewlet.update()
    return viewlet.render()

```

Rendering viewlets with accurate layout

Default viewlet managers render viewlets as HTML code string concatenation, in the order of appearance. This is unsuitable to build complex layouts.

Below is an example which defines master viewlet *HeaderViewlet* which will place other viewlets into the manually tuned HTML markup below.

theme/browser/header.py:

```

from Acquisition import aq_inner

# Use template files with acquisition support
from Products.Five.browser.pagetemplatefile import ViewPageTemplateFile

# Import default Plone viewlet classes
from plone.app.layout.viewlets import common as base

# Import our customized viewlet classes
# This is important as the header.py file will ignore much of the settings
# inside the configure.zcml file describing the affected viewlets. Without
# creating this file, your viewlets will render with Plone's default settings,
# which will result in your custom changes being ignored.
import plonetheme.something.browser.common as something

def render_viewlet(factory, context, request):
    """ Helper method to render a viewlet """

    context = aq_inner(context)
    viewlet = factory(context, request, None, None).__of__(context)
    viewlet.update()
    return viewlet.render()

class HeaderViewlet(base.ViewletBase):
    """ Render header with special markup.

    Though we render viewlets internally we not inherit from the viewlet manager,
    since we do not offer the option for the site manager or integrator
    shuffle viewlets - they are fixed to our templates.
    """

    index = ViewPageTemplateFile('header_items.pt')

    def update(self):

        base.ViewletBase.update(self)

        # Dictionary containing all viewlets which are rendered inside this viewlet.
        # This is populated during render()
        self.subviewlets = {}

    def renderViewlet(self, viewlet_class):
        """ Render one viewlet

        @param viewlet_class: Class which manages the viewlet
        @return: Resulting HTML as string
        """
        return render_viewlet(viewlet_class, self.context, self.request)

    def render(self):

        # Customized viewlet
        self.subviewlets["logo"] = self.renderViewlet(something.SomethingLogoViewlet)

        # Customized viewlet
        self.subviewlets["sections"] = self.renderViewlet(something.
        ↪SomethingGlobalSectionsViewlet)

```

```

    # Base Plone viewlet
    self.subviewlets["search"] = self.renderViewlet(base.SearchBoxViewlet)

    # Customized viewlet
    self.subviewlets["site_actions"] = self.renderViewlet(something.
↪SiteActionsViewlet)

    # Call template to perform rendering
    return self.index()

```

theme/browser/header_items.pt

```

<header>
  <div id="logo">
    <div tal:replace="structure view/subviewlets/logo" />
  </div>

  <nav>
    <div tal:replace="structure view/subviewlets/sections" />
  </nav>

  <div id="search">
    <div tal:replace="structure view/subviewlets/search" />
    <div id="actions">
      <div tal:replace="structure view/subviewlets/site_actions" />
    </div>
  </div>
</header>

```

theme/browser/configure.zcml

```

<configure xmlns="http://namespaces.zope.org/zope"
            xmlns:browser="http://namespaces.zope.org/browser"
            xmlns:plone="http://namespaces.plone.org/plone"
            xmlns:zcml="http://namespaces.zope.org/zcml"
            >

    <!--

        Public localizable site header

        See viewlets.xml for order/hidden

    -->

    <!-- Changes class and provides attributes to work with our changes -->
    <browser:viewletManager
        name="plone.portalheader"
        provides=".interfaces.ISomethingHeader"
        permission="zope2.View"
        class=".header.HeaderViewlet"
        layer=".interfaces.IThemeSpecific"
    />

    <!-- Site actions-->
    <browser:viewlet
        name="plonetheme.something.site_actions"
        class=".common.SiteActionsViewlet"
    />

```

```

        permission="zope2.View"
        template="templates/site_actions.pt"
        layer=".interfaces.IThemeSpecific"
        allowed_attributes="site_actions"
        manager=".interfaces.ISomethingHeader"
    />

    <!-- The logo; even though we include the template attribute, it will be ignored.
         Needs to be set again in common.py -->
    <browser:viewlet
        name="plonetheme.something.logo"
        class=".common.SomethingLogoViewlet"
        permission="zope2.View"
        layer=".interfaces.IThemeSpecific"
        template="templates/logo.pt"
        manager=".interfaces.ISomethingHeader"
    />

    <!-- Searchbox -->
    <browser:viewlet
        name="plone.searchbox"
        for="*"
        class="plone.app.layout.viewlets.common.SearchBoxViewlet"
        permission="zope2.View"
        template="templates/searchbox.pt"
        layer=".interfaces.IThemeSpecific"
        manager=".interfaces.ISomethingHeader"
    />

    <!-- First level navigation; even though we include the template attribute, it
    ↪will be ignored.
         Needs to be set again in common.py -->
    <browser:viewlet
        name="plonetheme.something.global_sections"
        for="*"
        class=".common.SomethingGlobalSectionsViewlet"
        permission="zope2.View"
        template="templates/sections.pt"
        layer=".interfaces.IThemeSpecific"
        manager=".interfaces.ISomethingHeader"
    />

</configure>

```

theme/browser/templates/portal_header.pt

```

<div id="portal-header">
    <div tal:replace="structure provider:plone.portalheader" />
</div>

```

theme/browser/interfaces.py code:

```

from plone.theme.interfaces import IDefaultPloneLayer
from zope.viewlet.interfaces import IViewletManager

class IThemeSpecific(IDefaultPloneLayer):
    """Marker interface that defines a Zope 3 browser layer.

```



```

    If you need to register a viewlet only for the
    "Something" theme, this interface must be its layer
    (in theme/viewlets/configure.zcml).
    """

class ISomethingHeader(IViewletManager):
    """Creates fixed layout for Plone header elements.
    """

```

We need to create this common.py file so we can tell Plone to render our custom templates for these viewlets. Without this piece in place, our viewlets will render with Plone defaults.

theme/browser/common.py code:

```

from Products.Five.browser.pagetemplatefile import ViewPageTemplateFile
from plone.app.layout.viewlets import common

# You may also use index in place of render for these subclasses

class SomethingLogoViewlet(common.LogoViewlet):
    render = ViewPageTemplateFile('templates/logo.pt')

class SomethingSiteActionsViewlet(common.SiteActionsViewlet):
    render = ViewPageTemplateFile('templates/site_actions.pt')

class SomethingGlobalSectionsViewlet(common.GlobalSectionsViewlet):
    render = ViewPageTemplateFile('templates/sections.pt')

```

Viewlets for one page only

Viewlets can be registered to one special page only using a marker interface. This allow loading a page specific CSS files.

- How to get a different look for some pages of a plone-site

<head> viewlets

You can register custom JavaScript or CSS files to HTML <head> section using viewlets.

Below is an head.pt which will be injected in <head>. This examples shows how to dynamically generate <script> elements.

```

<script type="text/javascript" tal:attributes="src view/getConnectScriptSource"></
↪script>
<script tal:replace="structure view/getInitScriptTag" />

```

Then you register it against viewlet manager `plone.app.layout.viewlets.interfaces.IHtmlHead` in `configure.zcml`

```

<browser:viewlet
    name="mfabrik.like.facebook-connect-head"
    class=".viewlets.FacebookConnectJavascriptViewlet"
    manager="plone.app.layout.viewlets.interfaces.IHtmlHead"
    template="facebook-connect-head.pt"
    layer="mfabrik.like.interfaces.IAddonInstalled"

```

```
permission="zope2.View"  
/>
```

viewlet.py code:

```
class FacebookConnectJavascriptViewlet(LikeButtonOnConnectFacebookBaseViewlet):  
    """ This will render Facebook JavaScript load in <head>.  
  
    <head> section is retrofitted only if the viewlet is enabled.  
  
    """  
  
    def getConnectScriptSource(self):  
        base = "http://static.ak.connect.facebook.com/connect.php/"  
        return base + self.getLocale()  
  
    def getInitScriptTag(self):  
        """ Get <script> which bootstraps Facebook stuff.  
        """  
        return '<script type="text/javascript">FB.init("%s");</script>' % self.  
↪ settings.api_key  
  
    def isEnabled(self):  
        """  
        @return: Should this viewlet be rendered on this page.  
        """  
        # Some logic based self.context here whether JavaScript should be included on_  
↪ this page or not  
        return True  
  
    def render(self):  
        """ Render viewlet only if it is enabled.  
  
        """  
  
        # Perform some condition check  
        if self.isEnabled():  
            # Call parent method which performs the actual rendering  
            return super(LikeButtonOnConnectFacebookBaseViewlet, self).render()  
        else:  
            # No output when the viewlet is disabled  
            return ""
```

Finding viewlets programmatically

Occasionally, you may need to get hold of your viewlets in python code, perhaps in tests. Since the availability of a viewlet is ultimately controlled by the viewlet manager to which it has been registered, using that manager is a good way to go

```
from zope.component import queryMultiAdapter  
from zope.viewlet.interfaces import IViewletManager  
  
from Products.Five.browser import BrowserView as View  
  
from my.package.tests.base import MyPackageTestCase
```

```

class TestMyViewlet(MyPackageTestCase):
    """ test demonstrates that registration variables worked
    """

    def test_viewlet_is_present(self):
        """ looking up and updating the manager should list our viewlet
        """
        # we need a context and request
        request = self.app.REQUEST
        context = self.portal

        # viewlet managers also require a view object for adaptation
        view = View(context, request)

        # finally, you need the name of the manager you want to find
        manager_name = 'plone.portalfooter'

        # viewlet managers are found by Multi-Adapter lookup
        manager = queryMultiAdapter((context, request, view), IViewletManager,
        ↪manager_name, default=None)
        self.assertIsNotNone(manager)

        # calling update() on a manager causes it to set up its viewlets
        manager.update()

        # now our viewlet should be in the list of viewlets for the manager
        # we can verify this by looking for a viewlet with the name we used
        # to register the viewlet in zcml
        my_viewlet = [v for v in manager.viewlets if v.__name__ == 'mypackage.
        ↪myviewlet']

        self.assertEqual(len(my_viewlet), 1)

```

Since it is possible to register a viewlet for a specific content type and for a browser layer, you may also need to use these elements in looking up your viewlet

```

from zope.component import queryMultiAdapter
from zope.viewlet.interfaces import IViewletManager
from Products.Five.browser import BrowserView as View
from my.package.tests.base import MyPackageTestCase

# this time, we need to add an interface to the request
from zope.interface import alsoProvides

# we also need our content type and browser layer
from my.package.content.mytype import MyType
from my.package.interfaces import IMyBrowserLayer

class TestMyViewlet(MyPackageTestCase):
    """ test demonstrates that zcml registration variables worked properly
    """

    def test_viewlet_is_present(self):
        """ looking up and updating the manager should list our viewlet
        """
        # our viewlet is registered for a browser layer. Browser layers
        # are applied to the request during traversal in the publisher. We

```

```
# need to do the same thing manually here
request = self.app.REQUEST
alsoProvides(request, IMyBrowserLayer)

# we also have to make our context an instance of our content type
content_id = self.folder.invokeFactory('MyType', 'my-id')
context = self.folder[content_id]

# and that's it. Everything else from here out is identical to the
# example above.
```

Poking viewlet registrations programmatically

Below is an example how one can poke viewlets registration for a Plone site.

```
from zope.component import getUtility
from plone.app.viewletmanager.interfaces import IViewletSettingsStorage

def fix_tinymce_viewlets(site):
    """
    Make sure TinyMCE viewlet is forced to be in Plone HTML <head> viewletmanager.

    For some reason, runnign in our viewlets.xml has no effect so we need to fix this_
    ↪by hand.
    """

    # Poke me like this: for i in storage._hidden["Isle of Back theme"].items():_
    ↪print i
    storage = getUtility(IViewletSettingsStorage)
    manager = "plone.htmlhead"
    skinname = site.getCurrentSkinName()

    # Force tinymce.configuration out of hidden viewlets in <head>
    hidden = storage.getHidden(manager, skinname)
    hidden = (x for x in hidden if x != u'tinymce.configuration')
    storage.setHidden(manager, skinname, hidden)
```

Layers

Description

Layers allow you to enable and disable views and other site functionality based on installed add-ons and themes.

Introduction

Layers allow you to activate different code paths and modules depending on the external configuration.

Examples:

- Code belonging to a theme is only active when that theme has been selected.
- Mobile browsing code is only active when the site is being browsed on a mobile phone.

Layers are marker interfaces applied to the `HTTPRequest` object. They are usually used in conjunction with *ZCML* directives to dynamically activate various parts of the configuration (theme files, add-on product functionality).

Layers ensure that only one add-on product can override the specific Plone instance functionality in your site at a time, while still allowing you to have possibly conflicting add-on products in your buildout and ZCML. Remember that multiple Plone site instances can share the same ZCML and code files.

Many ZCML directives take the optional `layer` parameter. See example, [resourceDirectory](#)

Layers can be activated when an add-on product is installed or a certain theme is picked.

For more information, read

- [Making components theme specific](#)
- [Browser Layer tutorial](#).
- [Zope 3 Developer Handbook, Skinning](#)

Using layers

Some ZCML directives (for example: `browser:page`) take a `layer` attribute.

If you have:

`plonetheme.yourthemename.interfaces.IThemeSpecific` layer defined in Python code

`YourTheme` product installed through add-on product installer on your site instance

then views and viewlets from your product can be enabled on the site instance using the following ZCML:

```
<!-- Site actions override in YourTheme -->
<browser:viewlet
    name="plone.site_actions"
    manager="plone.app.layout.viewlets.interfaces.IPortalHeader"
    class=".siteactions.SiteActionsViewlet"
    layer="plonetheme.yourthemename.interfaces.IThemeSpecific"
    permission="zope2.View"
/>
```

Unconditional overrides

If you want to override a view or a viewlet unconditionally for all sites without the add-on product installer support you need to use `overrides.zcml`.

Creating a layer

Theme layer

Theme layers can be created via the following steps:

1. Subclass an interface from `IDefaultPloneLayer`:

```
from plone.theme.interfaces import IDefaultPloneLayer

class IThemeSpecific(IDefaultPloneLayer):
    """Marker interface that defines a Zope 3 skin layer bound to a Skin
```

```
Selection in portal_skins.  
If you need to register a viewlet only for the "YourSkin"  
skin, this is the interface that must be used for the layer attribute  
in YourSkin/browser/configure.zcml.  
"""
```

2. Register it in ZCML. The name must match the theme name.

```
<interface  
  interface=".interfaces.IThemeSpecific"  
  type="zope.publisher.interfaces.browser.IBrowserSkinType"  
  name="SitsSkin"  
/>
```

3. Register and set your theme as the default theme in `profiles/default/skins.xml`. Theme layers require that they are set as the default theme and not just activated on your Plone site. Example:

```
<object name="portal_skins" allow_any="False" cookie_persistence="False"  
  default_skin="SitsSkin">  
  
  <!-- define skins-based folder objects here if any -->  
  
  <skin-path name="SitsSkin" based-on="Plone Default">  
    <layer name="plone_skins_style_folder_name"  
      insert-before="*" />  
  </skin-path>  
  
</object>
```

Add-on layer for clean extensions

An add-on product layer is enabled when an add-on product is installed. Since one Zope application server may contain several Plone sites, you need to keep enabled code paths separate by using add-on layers - otherwise all views and viewlets apply to all sites in one Zope application server.

- You can enable views and viewlets specific to functional add-ons.
- Unlike theme layers, add-on layers depend on the activated add-on products, not on the selected theme.

An add-on layer is a marker interface which is applied on the *HTTP request object* by Plone core logic.

First create an *interface* for your layer in your `.product.interfaces.py`:

```
""" Define interfaces for your add-on.  
"""  
  
import zope.interface  
  
class IAddOnInstalled(zope.interface.Interface):  
    """ A layer specific for this add-on product.  
  
    This interface is referred in browserlayer.xml.  
  
    All views and viewlets register against this layer will appear on  
    your Plone site only when the add-on installer has been run.  
    """
```

You then need to refer to this in the `profile/default/browserlayer.xml` file of your add-on installer *setup profile*:

```
<layers>
  <layer
    name="your.product"
    interface="your.product.interfaces.IAddOnInstalled"
  />
</layers>
```

Note: The add-on layer registry is persistent and stored in the database. The changes to add-on layers are applied only when add-ons are installed or uninstalled.

More information

- <https://pypi.python.org/pypi/plone.browserlayer>

Add-on layer for changing existing behavior

You can also use layers to modify the behavior of plone or another Add-on.

To make sure that your own view is used, your Layer must be more specific than the layer where original view is registered.

For example, some `z3cform` things register their views on the `IPloneFormLayer` from `plone.app.z3cform.interfaces`.

If you want to override the `ploneform-macros` view that is registered on the `IPloneFormLayer`, your own Layer must be a subclass of `IPloneFormLayer`.

If a view does not declare a specific Layer, it becomes registered on the `IDefaultBrowserLayer` from `zope.publisher.interfaces.browser.IDefaultBrowserLayer`.

Manual layers

Apply your layer to the `HTTPRequest` in the `before_traverse` hook or before you call the code which looks up the interfaces.

Choosing skin layer dynamically 1: http://blog.fourdigits.nl/changing-your-plone-theme-skin-based-on-the-objects-portal_type

Choosing skin layer dynamically 2: <http://code.google.com/p/plonegomobile/source/browse/trunk/gomobile/gomobile.mobile/gomobile/mobile/monkeypatch.py>

See the `plone.app.z3cform.z2` module.

In the example below we turn on a layer for the request which is later checked by the rendering code. This way some pages can ask for special View/Viewlet rendering.

Example:

```
# Defining layer

from zope.publisher.interfaces.browser import IBrowserRequest

class INoHeaderLayer(IBrowserRequest):
    """ When applied to HTTP request object, header animations or images are not_
    ↪ rendered on this.
```

```
If this layer is on request do not render header images.
This allows uncluttered editing of header animations and images.
"""

# Applying layer for some requests (manually done in view)
# The browser page which renders the form
class EditHeaderAnimationsView(FormWrapper):

    form = HeaderCRUDForm

    def __call__(self):
        """ """

        # Signal viewlet layer that we are rendering
        # edit view for header animations and it is not meaningful
        # to try to render the big animation on this page
        zope.interface.alsoProvides(self.request, INoHeaderLayer)

        # Render the edit form
        return FormWrapper.__call__(self)
```

Troubleshooting instructions for layers

- Check that your view or whatever is working without a layer assigned (globally);
- Check that `configure.zcml` has a layer entry. Put some garbage to trigger a syntax error in `configure.zcml` to make sure that it is being loaded;
- Add-on layer: check that `profiles/default/browserlayer.xml` has a matching entry with a matching name;
- Theme layer: if it's a theme layer, check that there is a matching `skins.xml` entry
- Check that layer name is correctly spelt in the view declaration.

Checking active layers

Layers are activated on the current request object

Example:

```
if INoHeaderLayer.providedBy(self.request):
    # The page has asked to suspend rendering of the header animations
    return ""
```

Active themes and add-on products

The `registered_layers()` method returns a list of all layers active on the site. Note that this is different to the list of layers which are applied on the current HTTP request object: the request object may contain manually activated layers.

Example:


```

from interfaces import IThemeSpecific
from plone.browserlayer.utils import registered_layers

if IThemeSpecific in registered_layers():
    # Your theme specific code
    pass
else:
    # General code
    pass

```

Getting active theme layer

Only one theme layer can be active at once.

The active theme name is defined in `portal_skins` properties. This name can be resolved to a theme layer.

Debugging active layers

You can check the activated layers from HTTP request object by looking at `self.request.__provides__.__iro__`. Layers are evaluated from zero index (highest priority) the last index (lowest priority).

Testing Layers

Plone testing tool kits won't register layers for you, you have to do it yourself somewhere in the boilerplate code:

```

from zope.interface import directlyProvides

directlyProvides(self.portal.REQUEST, IThemeLayer)

```

Tutorial: Overriding Viewlets

This tutorial¹ describes two simple examples of overriding viewlets. To learn more about views and viewlets, see the Developer Manual section on [Views and Viewlets](#)

Overriding the Logo

In this example, we override the logo for the site. I assume you have a theme product named `my.theme` with an `IThemeSpecific` interface.

1. Create an entry in `browser/configure.zcml` of your theme to override the viewlet.:

```

<browser:viewlet
    name="plone.logo"
    manager="plone.app.layout.viewlets.interfaces.IPortalHeader"
    class="plone.app.layout.viewlets.common.LogoViewlet"
    template="logo.pt"
    layer=".interfaces.IThemeSpecific"
    permission="zope2.View"
/>

```

¹ <https://plone.org/author/spanky>

2. Create a template file named `logo.pt` inside the browser directory that displays your logo image. It could contain something as simple as this.:

```
<div>
  <a metal:define-macro="portal_logo"
    accesskey="1"
    tal:attributes="href view/navigation_root_url"
    id:n:domain="plone">
    <img src="" tal:attributes="src string:${context/portal_url}/
    ↪++resource++my.theme.images/my_logo.png" alt="some alternative text" /></a>
</div>
```

3. Add your logo image to the browser/images directory of your theme. In this example, `++resource++my.theme.images/my_logo.png` points to a file named `my_logo.png` inside the theme's browser/images resource directory.

Overriding the Title

In this example we override the view class associated with the title viewlet. I assume you have a theme product with an `IThemeSpecific` interface.

1. Create an entry in `browser/configure.zcml` of your theme to override the view class.:

```
<browser:viewlet
  name="plone.htmlhead.title"
  manager="plone.app.layout.viewlets.interfaces.IHtmlHead"
  class=".common.TitleViewlet"
  layer=".interfaces.IThemeSpecific"
  permission="zope2.View"
/>
```

2. Create a class named `TitleViewlet` inside `browser/common.py` of your theme containing code to return the appropriate title.:

```
class TitleViewlet (ViewletBase):

    def update(self):
        # do any setup you need

    def index(self):
        ...
        return the appropriate title
```

Discussion

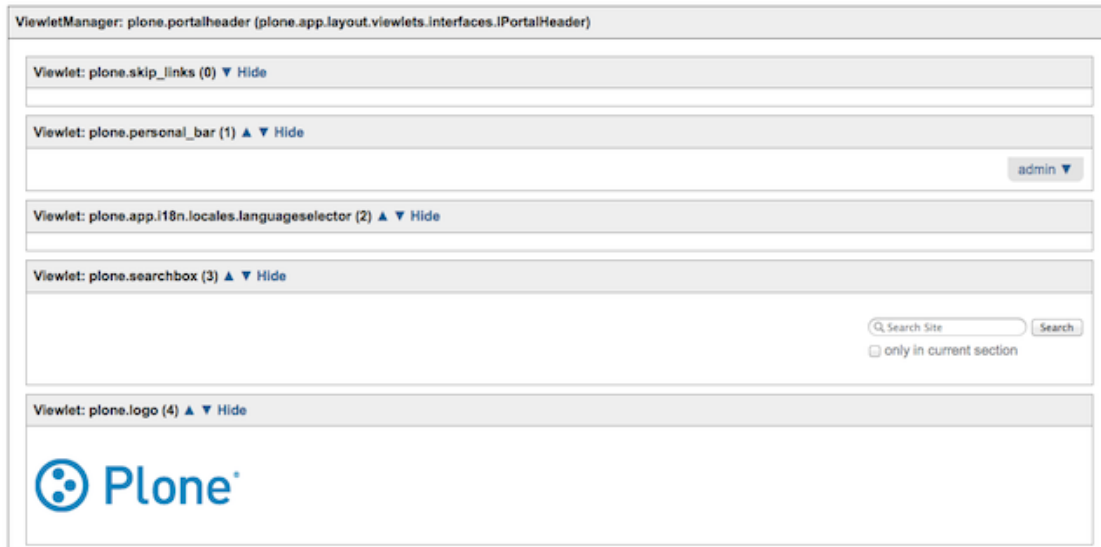
Overriding the logo

To override a viewlet in Plone, you need to know which viewlet to override. Using `@@manage-viewlets` is helpful here. It shows you all the viewlet managers on a page and the viewlets they contain.

You can add `/@@manage-viewlets` to any url in your site and see the active viewlets there. Something like:

```
http://localhost:8080/Plone/@@manage-viewlets
```

Using this shows us that the logo is in the `plone.logo` Viewlet within the `plone.portalheader` ViewletManager



Viewlets are defined in the `plone/app/layout/viewlets/configure.zcml` file within the eggs area of your buildout. Looking inside that `configure.zcml` file we see:

```
<!-- The logo -->
<browser:viewlet
    name="plone.logo"
    manager=".interfaces.IPortalHeader"
    class=".common.LogoViewlet"
    permission="zope2.View"
/>
```

Here's our overriding entry from above to compare:

```
<browser:viewlet
    name="plone.logo"
    manager="plone.app.layout.viewlets.interfaces.IPortalHeader"
    class="plone.app.layout.viewlets.common.LogoViewlet"
    template="templates/logo.pt"
    layer=".interfaces.IThemeSpecific"
    permission="zope2.View"
/>
```

The name is the same as the item we are overriding. Notice that we give the full path to the manager, and that we are reusing the class. We also declare the name and location of our overriding template file, use our theme's interface, and set a permission.

Overriding the title

Here is `TitleViewlet` from `plone.app.layout`. It has the page title on the left and the portal title on the right, with an emdash in between.:

```
class TitleViewlet (ViewletBase):
    index = ViewPageTemplateFile('title.pt')

    def update(self):
        portal_state = getMultiAdapter((self.context, self.request),
                                       name=u'plone_portal_state')
```

```
context_state = getMultiAdapter((self.context, self.request),
                                name=u'plone_context_state')
page_title = escape(safe_unicode(context_state.object_title()))
portal_title = escape(safe_unicode(portal_state.navigation_root_title()))
if page_title == portal_title:
    self.site_title = portal_title
else:
    self.site_title = u"%s &mdash; %s" % (page_title, portal_title)
```

Here is an example for comparison that switches page title and portal title, and separates them with a pipe. The only differences are on the last line.:

```
class TitleViewlet(ViewletBase):
    index = ViewPageTemplateFile('title.pt')

    def update(self):
        portal_state = getMultiAdapter((self.context, self.request),
                                        name=u'plone_portal_state')
        context_state = getMultiAdapter((self.context, self.request),
                                        name=u'plone_context_state')
        page_title = escape(safe_unicode(context_state.object_title()))
        portal_title = escape(safe_unicode(portal_state.navigation_root_title()))
        if page_title == portal_title:
            self.site_title = portal_title
        else:
            self.site_title = u"%s | %s" % (portal_title, page_title)
```

More information about the title tag can be found at the [HTML Head Title](#) page which is part of the [Plone Theme Reference](#).

Content management

In Plone, User editable objects are content objects. Content objects have different flavours depending on their type.

Examples:

- Pages (ATDocument class)
- Images (ATImage class)
- News items
- Events
- etc.

There are two different subsystems to create content objects in Plone

- Archetypes (Plone 2.x and Plone 3.x)
- Dexterity (Plone 4.x and Plone 3.x)

Creating objects

Description

Creating and controlling creation of Plone content items programmatically.

Creating content objects

Permission-aware way (Dexterity)

These instructions apply for *Dexterity* content types.

Example:

```
from plone.dexterity.utils import createContentInContainer

# Factory-type information id is the same as in types.xml
# optionally you can set checkConstraints=False to skip permission checks
item = createContentInContainer(folder, "your.app.dexterity.fti.information",
    title=title)
```

Permission-aware way (Archetypes and Dexterity)

`invokeFactory()` is available on all folderish content objects. `invokeFactory()` calls the `portal_factory` persistent utility to create content item.

Example:

```
def createResearcherById(folder, id):
    """ Create one researcher in a folder based on its X id.

    @param id: X id of the researcher

    @returns: Newly created researcher
    """

    # Call X REST service to get JSON blob for this researcher
    # Note: queryById parses JSON back to Python to do some sanity checks for it
    index = XPeopleIndex()
    oraData = index.queryById(id)

    # Need to have temporary id
    id = str(random.randint(0, 99999999))

    folder.invokeFactory("XResearcher", id)
    content = folder[id]

    # XResearcher stores its internal data as JSON
    json_data = json.dumps(oraData)
    content.setXData(json_data)

    # Will finish Archetypes content item creation process,
    # rename-after-creation and such
    content.processForm()

    return content
```

Example (from unit tests):

```
self.loginAsPortalOwner()
self.portal.invokeFactory("Folder", "folder")
```

```
self.portal.folder.invokeFactory("Folder", "subfolder")
self.portal.folder.subfolder.invokeFactory("Document", "doc")
```

`invokeFactory()` will raise an `Unauthorized` exception if the logged-in user does not have permission to create content in the folder (lacks type specific creation permission and Add portal content permissions). This exception can be imported as follows:

```
from Products.Archetypes.exceptions import AccessControl_Unauthorized
```

Note: If the content class has `_at_rename_after_creation = True` (Archetypes-based content) the next call to `obj.update()` (edit form post) will automatically generate a friendly id for the object based on the title of the object.

Bypassing permissions when creating content item

If you need to have special workflows where you bypass the workflow and logged in users when creating the content item, do as follows:

```
def construct_without_permission_check(folder, type_name, id, *args, **kwargs):
    """ Construct a new content item bypassing creation and content add permissions_
    ↪ checks.

    @param folder: Folderish content item where to place the new content item
    @param type_name: Content type id in portal_types
    @param id: Traversing id for the new content
    @param args: Optional arguments for the construction (will be passed to the_
    ↪ creation method if the type has one)
    @param kwargs: Optional arguments for the construction (will be passed to the_
    ↪ creation method if the type has one)
    @return: Reference to newly created content item
    """

    portal_types = getToolByName(folder, "portal_types")

    # Get this content type definition from content types registry
    type_info = portal_types.getTypeInfo(type_name)

    # _constructInstance takes optional *args, **kw parameters too
    new_content_item = type_info._constructInstance(folder, id)

    # Return reference to justly created content
    return new_content_item
```

Note: The function above only bypasses the content item construction permission check. It does not bypass checks for setting field values for initially created content.

There is also an alternative way:

```
# Note that by default Add portal member permissions
# is only for the owner, so we need to by pass it here
from Products.CMFPlone.utils import _createObjectByType
_createObjectByType("YourContentType", folder, id)
```

Manual friendly id generation

If you are creating Plone objects by hand e.g. in a batch job and Plone automatic id generation does not kick in, you can use the following example to see how to create friendly object ids manually:

```
from zope.component import getUtility
from plone.i18n.normalizer.interfaces import IIDNormalizer

import transaction

def createResearcherById(folder, id):
    """ Create one researcher in a folder based on its ORA id.

    @param id: X id of the researcher

    @returns: Newly created researcher
    """

    # Call X REST service to get JSON blob for this researcher
    # Note: queryById parses JSON back to Python to do some sanity checks for it
    index = XPeopleIndex()

    # Need to have temporary id
    id = str(random.randint(0, 99999999))

    folder.invokeFactory("XResearcher", id)
    content = folder[id]

    # XXX: set up content item data

    # Will finish Archetypes content item creation process,
    # rename-after-creation and such
    content.processForm()

    # make _p_jar on content
    transaction.savepoint(optimistic=True)

    # Need to perform manual normalization for id,
    # as we don't have title available during the creation time
    normalizer = getUtility(IIDNormalizer)
    new_id = normalizer.normalize(content.Title())

    if new_id in folder.objectIds():
        raise RuntimeError("Item already exists:" + new_id + " in " + folder.absolute_
↪ url())

    content.aq_parent.manage_renameObject(id, new_id)

    return content
```

PortalFactory

PortalFactory (only for Archetypes) creates the object in a temporary folder and only moves it to the real folder when it is first saved.

Note: To see if content is still temporary, use `portal_factory.isTemporary(obj)`.

Restricting creating on content types

Plone can restrict which content types are available for creation in a folder via the *Add...* menu.

Restricting available types per content type

`portal_types` defines which content types can be created inside a folderish content type. By default, all content types which have the `global_allow` property set can be added.

The behavior can be controlled with `allowed_content_types` setting.

- You can change it through the `portal_types` management interface.
- You can change it in your add-on installer *GenericSetup* profile.

Example for *Dexterity content type*. The file would be something like `profiles/default/types/yourcompany.app.typeid.xml`:

```
<!-- List content types we allow here -->
<property name="filter_content_types">True</property>
<property name="allowed_content_types">
  <element value="yourcompany.app.courseinfo" />
</property>
<property name="allow_discussion">False</property>
```

Example for *Archetypes content*. The file would be something like `profiles/default/types/YourType.xml`:

```
<property name="filter_content_types">True</property>

<property name="allowed_content_types">
  <element value="YourContentTypeName" />
  <element value="Image" />
  <element value="News Item" />
  ...
</property>
```

Restricting available types per folder instance

In the UI, you can access this feature via the *Add...* menu *Restrict* option.

Type contraining is managed by the `ATContentTypes` product:

- <https://github.com/plone/Products.ATContentTypes/blob/master/Products/ATContentTypes/lib/constrainttypes.py>

Example:

```
# Set allowed content types
from Products.ATContentTypes.lib import constrainttypes

# Enable contstraining
```



```
folder.setConstrainTypesMode(constraintypes.ENABLED)

# Types for which we perform Unauthorized check
folder.setLocallyAllowedTypes(["ExperienceEducator"])

# Add new... menu listing
folder.setImmediatelyAddableTypes(["ExperienceEducator"])
```

You can also override the `constraintypes` accessor method to have programmable logic regarding which types are addable and which not.

Other restrictions

See this discussion thread:

- <http://plone.293351.n2.nabble.com/Folder-constraints-not-applicable-to-custom-content-types-tp6073100p6074327.html>

Creating OFS objects

Zope has facilities for basic folder and contained objects using the `IObjectManager` definition subsystem. You do not need to work with raw objects unless you are doing your custom lightweight, Plone-free, persistent data.

Object construction life cycle

Note: The following applies to Archetypes-based objects only. The process might be different for Dexterity-based content.

Archetypes content construction has two phases:

1. The object is created using a `?createType=` URL or a `Folder.invokeFactory()` call. If `createType` is used then the object is given a temporary id. The object has an “in creation” flag set.
2. The object is saved for the first time and the final id is generated based on the object title. The object is renamed. The creation flag is cleared.

You are supposed to call either `object.unmarkCreationFlag()` or `object.processForm()` after content is created manually using `invokeFactory()`.

`processForm()` will perform the following tasks:

- unmarks creation flag;
- renames object according to title;
- reindexes object;
- invokes the `after_creation` script and fires the `ObjectInitialized` event.

If you don't want some particular step to be executed, study `Archetypes/BaseObject.py` and call only what you really want. But unless `unmarkCreationFlag()` is called, the object will behave strangely after the first edit.

Rename after creation

To prevent the automatic rename on the first through-the-web save, add the following attribute to your class:

```
_at_rename_after_creation = False
```

Factory type information

Factory type information (FTI) is responsible for content creation in the portal. It is independent from content type (Archetypes, Dexterity) subsystems.

Warning: The FTI codebase is old (updated circa 2001). Useful documentation might be hard to find.

FTI is responsible for:

- Which function is called when new content type is added;
- icons available for content types;
- creation views for content types;
- permission and security;
- whether discussion is enabled;
- providing the `factory_type_information` dictionary. This is used elsewhere in the code (often in `__init__.py` of a product) to set the initial values for a *ZODB Factory Type Information* object (an object in the `portal_types` tool).

See:

- [FTI source code](#).
- [Scriptable Types Information HOW TO](#)
- [Notes Zope types mechanism](#)

Content does not show in *Add* menu, or *Unauthorized* errors

These instructions are for Archetypes content to debug issues when creating custom content types which somehow fail to become creatable.

When creating new content types, many things can silently fail due to human errors in the complex content type setup chain and security limitations. The consequence is that you don't see your content type in the *Add* drop-down menu. Here are some tips for debugging.

- Is your product broken due to Python import time errors? Check the Management Interface: *Control panel* -> *Products*. Turn on Zope debugging mode to trace import errors.
- Have you rerun the quick installer (`GenericSetup`) after creating/modifying the content type?
- Do you have a correct *Add* permission for the product? Check `__init__.py` `ContentInit()` call.
- Does it show up in the portal factory? Check the Management Interface: *portal_factory* and `factorytool.xml`.
- Is it correctly registered as a portal type and implicitly addable? Check the Management Interface: *portal_types*. Check `default/profiles/type/yourtype.xml`.

- Does it have correct product name defined? Check the Management Interface: *portal_types*.
- Does it have a proper factory method? Check Management Interface: *portal_types*. Check Zope logs for `_queryFactory` and import errors.
- Does it register itself with Archetypes? Check the Management Interface: *archetypes_tool*. Make sure that you have `ContentInit` properly run in your `__init__.py`. Make sure that all modules having Archetypes content types defined and `registerType()` call are imported in `__init__.py`.

Link to creation page

- The *Add...* menu contains links for creating individual content types. Copy the URLs that you see there.
- If you want to the user to have a choice about which content type to create, you can link to `/folder_factories` page. (This is also the creation page when JavaScript is disabled).

Populating folder on creation

Archetypes have a hook called `initializeArchetype()`. Your content type subclass can override this.

Example:

```
class LandingPage(folder.ATFolder):
    """Landing page"""

    def initializeArchetype(self, **kwargs):
        """
        Prepopulate folder during the creation.

        Create five subfolders of "BigBlock" type, with title and id preset.
        """
        folder.ATFolder.initializeArchetype(self, **kwargs)

        for i in range(0, 5):
            id = "container" + str(i)
            self.invokeFactory("BigBlock", id, title="Big block " + str(i+1))
            item = self[id]

            # Clear creation flag
            item.markCreationFlag()
```

Creating content from PloneFormGen

PloneFormGen is a popular add-on for Plone.

Below is a snippet for a Custom Script Adapter which allows to create content straight out of PloneFormGen in the *pending* review state (it is not public and will appear in the review list):

```
# Folder id where we create content is "directory" under site root
target = context.portal_url.getPortalObject()["directory"]

# The request object has an dictionary attribute named
# form that contains the submitted form content, keyed
# by field name
form = request.form
```

```
# We need to engineer a unique ID for the object we're
# going to create. If your form submit contained a field
# that was guaranteed unique, you could use that instead.
from DateTime import DateTime
uid = str(DateTime().millis())

# We use the "invokeFactory" method of the target folder
# to create a content object of type "Document" with our
# unique ID for an id and the form submission's topic
# field for a title.

# Field id have been set in Form Folder Contents view,
# using rename functionality
target.invokeFactory("Document", id=uid,
                      title=form['site-name'],
                      description=form['site-description'],
                      remoteUrl=form["link"]
                      )

# Find our new object in the target folder
obj = target[uid]

# Trigger rename-after-creation behavior
# where actual id is generated from the title
obj.processForm()

# Make item to pending state
portal_workflow = context.portal_workflow
portal_workflow.doActionFor(obj, "submit")
```

More info:

- <https://plone.org/products/ploneformgen/documentation/how-to/creating-content-from-pfg>
- <https://plone.org/products/ploneformgen/documentation/how-to/creating-content-from-pfg>

Creating content using Generic Setup

Purpose

You want your product to create default content in the site. (For example, because you have a product which adds a new content type, and you want to create a special folder to put these items in.)

You could do this programmatically, but if you don't want anything fancy (see "Limitations" below), Generic Setup can also take care of it.

Step by step

- In your product's profiles/default folder, create a directory called `structure`.
- To create a top-level folder with id `my-folder-gs-created`, add a directory of that name to the `structure` folder.
- Create a file called `.objects` in the `structure` directory
- Create a file called `.properties` in the `my-folder-gs-created` directory

- Create a file called `.preserve` in the `structure` directory
- `.objects` registers the folder to be created:

```
my-folder-gs-created, Folder
```

- `.properties` sets properties of the folder to be created:

```
[DEFAULT]
description = Folder for imported Projects
title = My folder (created by generic setup)
```

- `.preserve` will make sure the folder isn't overwritten if it already exists:

```
my-folder-gs-created
```

Limitations

- This will only work for Plone's own content types
- Items will be in their initial workflow state

If you want to create objects of a custom content type, or manipulate them more, you'll have to write a setuphandler. See below under "Further Information".

Troubleshooting

I don't see titles in the navigation, only ids

You may notice that the new generated content's title appears to be set to its id. In this case, the catalog needs to be updated. You can do this through the Management Interface, in `portal_catalog`.

You could automate this process by adding a GS import step in `configure.zcml`, which looks like this:

```
<genericsetup:importStep
    name="my.policy_updateCatalog"
    title="Update catalog"
    description="After creating content (from profiles/default/structure), the_
↪catalog needs to be updated."
    handler="my.policy.setuphandlers.updateCatalog">
    <depends name="content"/>
</genericsetup:importStep>
```

This is the preferred way to define dependencies for import profiles: The import step declares its dependency on the content import step. 'content' is the name for the step which creates content from `profiles/default/structure`. You could then add a method which updates the catalog in the product's `setuphandlers.py`:

```
def updateCatalog(context, clear=True):
    portal = context.getSite()
    logger = context.getLogger('my.policy_updateCatalog')
    logger.info('Updating catalog (with clear=%s) so items in profiles/default/
↪structure are indexed...' % clear)
    catalog = portal.portal_catalog
    err = catalog.refreshCatalog(clear=clear)
    if not err:
        logger.info('...done.')
```

```
else:
    logger.warn('Could not update catalog.')
```

Further information

- Original manual: <http://vanrees.org/weblog/creating-content-with-genericsetup>
- If you want to do things like workflow transitions or setting default views after creating, read <http://keeshink.blogspot.de/2011/05/creating-plone-content-when-installing.html>

Listing objects

Description

How to programmatically generate folder listings in Plone.

Introduction

Plone has several methods of getting the list of folder items, depending on whether:

- you want to get all items, or only items visible for the currently logged in user;
- you want to get hold of the item objects themselves or indexed metadata (the latter is faster);
- you want to get Plone's contentish items only (`contentItems`) or Zope 2 management objects too (`objectIds`); the latter covers various site utilities found in the portal root and otherwise hidden magical items.

Special attention must be paid also to object ids. Zope locates all objects by traversing the site graph using ids. The id mapping is usually a property of a *parent* object, not the child. Thus most of the listing methods tend to return `(id, object)` tuples instead of plain objects.

Ensuring that the content item is a folder

All Plone folderish content types provide the `IFolderish` interface. Check that this is present to make sure that a content item is a folder, and that `contentItems()` and the other methods are available:

```
from Products.CMFCore.interfaces import IFolderish

def recurse_all_content(portal):

    output = StringIO()

    def recurse(context):
        """ Recurse through all content on Plone site """

        print >> output, "Recurring to item:" + str(context)

        # Make sure that we recurse to real folders only,
        # otherwise contentItems() might be acquired from higher level
        if IFolderish.providedBy(context):
            for id, item in context.contentItems():
```

```

        recurse(item)

recurse(portal)

return output

```

Getting all content objects inside a folder

The `contentItems` method is defined in `CMFCore/PortalFolder.py`. From Plone 4 and later, you can also use `folder.items()` instead (this applies to the whole section below). See source code for details, e.g. filtering and other forms of listing.

Querying folder through catalog

These methods apply for real folders, and not for collections.

Getting indexed objects

This is a faster method. `portal_catalog` must be up-to-date for the folder. This will return *brain objects*:

```
brains = folder.getFolderContents()
```

Getting full objects

```
items = folder.contentItems() # return Python list of children object tuples (id,
↪object)
```

Warning: The `contentItems()` call may be costly, since it will return the actual content objects, not the indexed metadata from the `portal_catalog`. You should avoid this method if possible.

Warning: `folder.contentItems()` returns all items regardless of the user security context.

Getting folder objects filtered

The `listFolderContents()` method retrieves the content objects from the folder. It takes `contentFilter` as an argument to specify filtering of the results. `contentFilter` uses the same syntax as `portal_catalog` queries, but does not support all the same parameters; e.g. `object_provides` is not supported. See the [Content-Filter class](#) for details.

Example:

```
# List all types in this folder whose portal_type is "CourseModulePage"
return self.listFolderContents(contentFilter={"portal_type" : "CourseModulePage"})
```

Warning: Security warning: `listFolderContents()` honors the currently logged-in user roles.

Warning: Performance warning: slow for large folders. Rather use `portal_catalog` and path-based queries to query items in a large folder.

Rules for filtering items

Plone applies some default rules for `listFolderContents()`

- `portal_properties.nav_tree_properties.metaTypesNotToQuery`: folders (large folders) don't generate listing.
- *default_page* are not listed.
- `portal_properties.nav_tree_properties`: meta types marked here do not appear in the listing.

Why does `folder_listing` not list my contents?

The site search settings (*Site Setup*→*Search*) modifies the way `folder_listing` works.

If you specify that you do not want to search objects of type *Page*, they will not appear in `folder_listing` anymore.

From [this thread](#).

`orderObjects()` to set a key for ordering the items in a particular folder

With Plone 4+ an adapter can be registered and used to apply a custom order to a particular folder: see `setOrdering`. The `DefaultOrdering` adapter allows a key to be set for a particular folder, and optionally to reverse the order. This can be adjusted via a method on the folder:

```
context.orderObjects(key="Title", reverse=True)
```

Note: Unlike the python `sort()` and `sorted()` methods, the key parameter expects an attribute, not a function.

Enforcing manual sort order

Below is an example of how to order content items by their manual sort order (the one you create via drag and drop on the contents tab):

```
from OFS.interfaces import IOrderedContainer

queried_objects = list(folder.listFolderContents())

def get_position_in_parent(obj):
    """
    Use IOrderedContainer interface to extract the object's manual ordering position
    """
```



```

parent = obj.aq_inner.aq_parent
ordered = IOrderedContainer(parent, None)
if ordered is not None:
    return ordered.getObjectPosition(obj.getId())
return 0

def sort_by_position(a, b):
    """
    Python list sorter cmp() using position in parent.

    Descending order.
    """
    return get_position_in_parent(a) - get_position_in_parent(b)

queried_objects = sorted(queried_objects, sort_by_position)

```

Getting object ids

If you need to get ids only, use the `objectIds()` method, or `keys()` in Plone 4. This is a fast method:

```

# Return a list of object ids in the folder
ids = folder.objectIds() # Plone 3 or older
ids = folder.keys()      # Plone 4 or newer

```

Warning: `objectIds()` and `keys()` will return ids for raw Zope 2 objects too, not just Plone content. If you call `objectIds()` on the portal root object, you will get objects like `acl_users`, `portal_workflow` etc ...

Getting non-contentish Zope objects

In some special cases, it is necessary to manipulate non-contentish Zope objects.

This listing method applies to all `OFS.Folder.Folder` objects, not just Plone content objects:

```

for id, item in folder.objectItems():
    # id is 8-bit string of object id in the folder
    # item is the object itself
    pass

```

Checking for the existence of a particular object id

If you want to know whether the folder has a certain item or not, you can use the following snippet.

Plone 4

Use `has_key`:

```

if folder.has_key("my-object-id"):
    # Exists
else:
    # Does not exist

```

Listing the folder items using `portal_catalog`

This should be your preferred method for querying folder items. `portal_catalog` searches are fast, because they return catalog brain objects instead of the real content objects (less database look ups).

Warning: Returned catalog brain data, such as `Title`, will be UTF-8 encoded. You need to call `brain["title"].decode("utf-8")` or similar on all text you want to extract from the data.

Simple example how to get all items in a folder:

```
# Get the physical path (includes Plone site name)
# to the folder
path = folder.getPhysicalPath()

# Convert getPhysicalPath() tuples result to
# slash separated string, which is used by ExtendedPathIndex
path = "/".join(path)

# This will fetch catalog brains.
# Includes also unpublished items, not caring about workflow state.
# depth = 1 means that subfolder items are not included

brains = context.portal_catalog(path={"query": path, "depth": 1})
```

Here's a complex example of how to perform various filtering operations, honouring some default Plone filtering rules. This example is taken from `Products.CMFPlone/skins/plone_scripts/getFolderContents`:

```
mtool = context.portal_membership
cur_path = '/'.join(context.getPhysicalPath())
path = {}

if not contentFilter:
    # The form and other are what really matters
    contentFilter = dict(getattr(context.REQUEST, 'form', {}))
    contentFilter.update(dict(getattr(context.REQUEST, 'other', {})))
else:
    contentFilter = dict(contentFilter)

if not contentFilter.get('sort_on', None):
    contentFilter['sort_on'] = 'getObjPositionInParent'

if contentFilter.get('path', None) is None:
    path['query'] = cur_path
    path['depth'] = 1
    contentFilter['path'] = path

show_inactive = mtool.checkPermission(
    'Access inactive portal content', context)

# Evaluate in catalog context because some containers override queryCatalog
# with their own unrelated method (Topics)
contents = context.portal_catalog.queryCatalog(
    contentFilter, show_all=1, show_inactive=show_inactive)
```

```

if full_objects:
    contents = [b.getObject() for b in contents]

if batch:
    from Products.CMFPlone import Batch
    b_start = context.REQUEST.get('b_start', 0)
    batch = Batch(contents, b_size, int(b_start), orphan=0)
    return batch

return contents

```

Count of content items

Counting items using `getFolderContents`

The least expensive call for this, if you have tens of items, is to call `len()` on the result of calling `getFolderContents()`, which is a `portal_catalog` based query:

```
items = len(self.getFolderContents())
```

Counting items using `contentItems`

Alternatively, if you know there are not many objects in in the folder, you can call `contentItems()` (or simply `items()` in Plone 4 or newer), as this will potentially wake fewer items than a complex catalog query.

Warning: Security: This method does not consider access rights.

Example (AT content class method):

```

def getMainImage(self):
    items = self.contentItems() # id, object tuples
    # "items = self.items()" in Plone 4 or newer
    if len(items) > 0:
        return items[1]

```

Navigational view URL

Plone has a special default navigation URL which is used in

- Folder listing
- Navigation tree

It is not necessarily the object URL itself (`/folder/item`), but can be e.g. `/folder/item/@@yourcustomview`

The view action URL must be configured in `portal_types` and separately enabled for the content type in `site_properties`.

For more information see

- http://stackoverflow.com/questions/12033414/change-link-in-contents-listing-for-custom-content-type#comment16065296_12033414

Custom folder listing

Here is an example how to create a view which will render a custom listing for a folder or a collection (ATTopic).

The view is called `ProductSummaryView` and it is registered with the name `productsummary`. This example is not suitable for your add-on product as is: you need to tailor it for your specific needs.

Warning: If you are going to call `item/getObject` on a catalog brain, it might cause excessive database load as it causes a new database query per object. Try use information available in the catalog or add more catalog indexes. To know more about the issue read about waking up database objects.

- First, let's register our view. We could limit content types for which the view is enabled by specifying `Products.ATContentTypes.interface.IATFolder` or `Products.ATContentTypes.interface.IATTopic` in the `for` attribute. Cf. the `configure.zcml` snippet below:

```
<browser:page
  for="*"
  name="productcardsummary"
  class=".productcardsummaryview.ProductCardSummaryView"
  template="productcardsummaryview.pt"
  allowed_interface=".productcardsummaryview.IProductCardSummaryView"
  permission="zope2.View"
/>
```

- Below is the example view code, named as `productcardsummaryview.py`:

```
from zope.interface import implements, Interface

from zope import schema

from Products.Five import BrowserView
from Products.CMFCore.utils import getToolByName

from Products.ATContentTypes.interface import IATTopic

# zope.18n message translator for your add-on product
from yourproduct.namespace import appMessageFactory as _

class IProductCardSummaryView(Interface):
    """ Allowed template variables exposed from the view.
    """

    # Item list as iterable Products.CMFPlone.PloneBatch.Batch object
    contents = schema.Object(Interface)

class ProductCardSummaryView(BrowserView):
    """
    List summary information for all product cards in the folder.

    Batch results.
    """
    implements(IProductCardSummaryView)
```

```

def query(self, start, limit, contentFilter):
    """ Make catalog query for the folder listing.

    @param start: First index to query

    @param limit: maximum number of items in the batch

    @param contentFilter: portal_catalog filtering dictionary with index ->
    ↪value pairs.

    @return: Products.CMFPlone.PloneBatch.Batch object
    """

    # Batch size
    b_size = limit

    # Batch start index, zero based
    b_start = start

    # We use different query method, depending on
    # whether we do listing for topic or folder
    if IATopic.providedBy(self.context):
        # ATTopic like content
        # Call Products.ATContentTypes.content.topic.ATTopic.queryCatalog()
    ↪method
        # This method handles b_start internally and
        # grabs it from HTTPRequest object
        return self.context.queryCatalog(contentFilter, batch=True, b_size=b_
    ↪size)
    else:
        # Folder or Large Folder like content
        # Call CMFPlone(/skins/plone_scripts/getFolderContents Python script
        # This method handles b_start parametr internally and grabs it from
    ↪the request object
        return self.context.getFolderContents(contentFilter, batch=True, b_
    ↪size=b_size)

def __call__(self):
    """ Render the content item listing.
    """

    # How many items is one one page
    limit = 3

    # What kind of query we perform?
    # Here we limit results to ProductCard content type
    filter = { "portal_type" : "ProductCard" }

    # Read the first index of the selected batch parameter as HTTP GET
    ↪request query parameter
    start = self.request.get("b_start", 0)

    # Perform portal_catalog query
    self.contents = self.query(start, limit, filter)

    # Return the rendered template (productcardsummaryview.pt), with content
    ↪listing information filled in

```

```
return self.index()
```

- Below is the corresponding page template skeleton `productcardsummaryview.pt`:

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
lang="en"
metal:use-macro="here/main_template/macros/master"
i18n:domain="yourproduct.namespace">
<body>
  <div metal:fill-slot="main">
    <tal:main-macro metal:define-macro="main">

      <div tal:replace="structure provider:plone.abovecontenttitle" />

      <h1 metal:use-macro="here/kss_generic_macros/macros/generic_title_view">
        Title or id
      </h1>

      <div tal:replace="structure provider:plone.belowcontenttitle" />

      <p metal:use-macro="here/kss_generic_macros/macros/generic_description_
↪view">
        Description
      </p>

      <div tal:replace="structure provider:plone.abovecontentbody" />

      <tal:listing define="batch view/contents">

        <tal:block tal:repeat="item batch">
          <div class="tileItem visualIEFloatFix vevent"
            tal:define="normalizeString nocall: context/plone_utils/
↪normalizeString;
                                item_url item/getURL|item/absolute_url;
                                item_id item/getId|item/id;
                                item_title_or_id item/pretty_title_or_id;
                                item_description item/Description;
                                item_type item/portal_type;
                                item_type_title item/Type;
                                item_type_class python: 'contenttype-' +
↪normalizeString(item_type);
                                item_modified item/ModificationDate;
                                item_created item/CreationDate;
                                item_wf_state item/review_
↪state|python: wtool.getInfoFor(item, 'review_state', '');
                                item_wf_state_class python:'state-' +
↪normalizeString(item_wf_state);
                                item_creator item/Creator;
                                item_start item/start/ISO|item/
↪StartDate|nothing;
                                item_end item/end/ISO|item/EndDate|nothing;
                                "
                                tal:attributes="class string:tileItem visualIEFloatFix_
↪vevent ${item_type_class}">

            <a href="#"
              tal:attributes="href item_url">
```

→ "

Making view available in the *Display...* menu

You need to add the `browser:menuItem` entry to make your view appear in the *Display...* menu from which folders and topics can choose the style of the display.

See *dynamic views*.

You need to add:

- `<browser:menuItem>` configuration directive with view id (e.g. `@@productsummary`)
- New properties to `Folder.xml` or `Topic.xml` so that the view becomes available

Preventing folder listing

If the users can access the content items they can usually also list them.

Here is a no-warranty hack how to prevent `folder_listing` if needed:

```
from zope.component import adapter
from ZPublisher.interfaces import IPubEvent, IPubAfterTraversal
from Products.CMFCore.utils import getToolByName
from AccessControl.unauthorized import Unauthorized
from zope.app.component.hooks import getSite

@adapter(IPubAfterTraversal)
def Protector(event):
    """ Protect anonymous users from access to folder_listing etc. """

    site = getSite()
    if not site:
        return

    ms = getToolByName(site, 'portal_membership')
    member = ms.getAuthenticatedMember()
    if not member.getUserName() == 'Anonymous User':
        return

    URL = event.request.URL
    if '/folder_' in URL:
        raise Unauthorized('unable to access folder listing')
```

Complex folder listings and filtering

The following example is for a complex folder listing view.

You can call view methods to return the listed items themselves and render the HTML in another view — this allows you to recycle this listing code.

The view does the various sanity checks that normal Plone item listings do:

- no meta items,
- no large folders,
- no default views,
- filter by active language,

- do not list items where you do not have the View permission,
- perform the listing on the parent container if the context itself is not folderish.

Example code:

```
class FolderListingView(BrowserView):
    """ Mobile folder listing helper view

    Use getItem() to get list of mobile folder listable items for
    automatically generated mobile folder listings (touch button list).
    """

    def getListingContainer(self):
        """ Get the item for which we perform the listing
        """
        context = self.context.aq_inner
        if IFolderish.providedBy(context):
            return context
        else:
            return context.aq_parent

    def getActiveTemplate(self):
        state = getMultiAdapter(
            (self.context, self.request),
            name=u'plone_context_state')
        return state.view_template_id()

    def getTemplateIdsNoListing(self):
        """
        @return: List of mobile-specific ids found from portal_properties where not_
        ↪to show folder listing
        """

        try:
            from gomobile.mobile.utilities import getCachedMobileProperties
            context = aq_inner(self.context)
            mobile_properties = getCachedMobileProperties(context, self.request)
        except:
            mobile_properties = None

        return getattr(mobile_properties, "no_folder_listing_view_ids", [])

    def filterItems(self, container, items):
        """ Apply mobile specific filtering rules

        @param items: List of context brains
        """

        # Filter out default content
        default_page_helper = getMultiAdapter(
            (container, self.request),
            name='default_page')

        portal_state = getMultiAdapter(
            (container, self.request),
            name='plone_portal_state')
```

```
# Active language
language = portal_state.language()

# Return the default page id or None if not set
default_page = default_page_helper.getDefaultPage(container)

security_manager = getSecurityManager()

meta_types_not_to_list = container.portal_properties.navtree_properties.
↳metaTypesNotToList

def show(item):
    """ Filter whether the user can view a mobile item.

    @param item: Real content object (not brain)

    @return: True if item should be visible in the listing
    """

    # Check from mobile behavior should we do the listing
    try:
        behavior = IMobileBehavior(item)
        appearInFolderListing = behavior.appearInFolderListing
    except TypeError:
        # Site root or some weird object, give up
        appearInFolderListing = True

    if not appearInFolderListing:
        # Default to appearing
        return False

    # Default page should not appear in the quick listing
    if item.getId() == default_page:
        return False

    if item.meta_type in meta_types_not_to_list:
        return False

    # Two letter language code
    item_lang = item.Language()

    # Empty string makes language netral content
    if item_lang not in ["", None]:
        if item_lang != language:
            return False

    # Note: getExcludeFromNav not necessarily exist on all content types
    if hasattr(item, "getExcludeFromNav"):
        if item.getExcludeFromNav():
            return False

    # Does the user have a permission to view this object
    if not security_manager.checkPermission(permissions.View, item):
        return False

    return True
```

```

    return [ i for i in items if show(i) == True ]

def constructListing(self):
    # Iterable of content items for the item listing
    items = []

    # Check from mobile behavior should we do the listing
    try:
        behavior = IMobileBehavior(self.context)
        do_listing = behavior.mobileFolderListing
    except TypeError:
        # Site root or some weird object, give up
        do_listing = False

    # Do listing by default, must be explicitly disabled
    if not do_listing:
        # No mobile behavior -> no mobile listing
        return None

    container = self.getListingContainer()

    # Do not list if already doing folder listing
    template = self.getActiveTemplate()
    print "Active template id:" + template
    if template in self.getTemplateIdsNoListing():
        # Listing forbidden by mobile rules
        return None

    portal_properties = getToolByName(container, "portal_properties")
    navtree_properties = portal_properties.navtree_properties
    if container.meta_type in navtree_properties.parentMetaTypesNotToQuery:
        # Big folder... listing forbidden
        return None

    state = container.restrictedTraverse('@@plone_portal_state')

    items = container.listFolderContents()

    items = self.filterItems(container, items)

    return items

def getItems(self):
    """
    @return: Iterable of content objects. Never return None.
    """
    items = self.constructListing()
    if items == None:
        return []
    return items

```

Empty listing view

Sometimes you want a show folder without listing its content. You can create a *dynamic view* in your add-on which is available from *Display...* menu.

Example `configure.zcml` bit

```
<browser:page
    name="empty-listing"
    for="Products.CMFCore.interfaces.IFolderish"
    permission="zope2.View"
    layer=".interfaces.IThemeSpecific"
    template="empty-listing.pt"
/>
```

Example `empty-listing.pt`

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:metal="http://xml.zope.org/namespaces/metal"
      xmlns:tal="http://xml.zope.org/namespaces/tal"
      xmlns:i18n="http://xml.zope.org/namespaces/i18n"
      i18n:domain="example.dexterityforms"
      metal:use-macro="context/main_template/macros/master">

    <metal:block fill-slot="content-title">
    </metal:block>

    <metal:block fill-slot="content-core">
    </metal:block>

</html>
```

Example `profiles/default/types/Folder.xml`

```
<?xml version="1.0"?>
<object name="Folder"
  xmlns:i18n="http://xml.zope.org/namespaces/i18n"
  i18n:domain="plone"
  meta_type="Factory-based Type Information with dynamic views" >
  <property name="view_methods" purge="False">
    <!-- We retrofit these new views for Folders in portal_types info -->
    <element value="empty_listing"/>
  </property>
</object>
```

Reinstall your add-on.

empty-listing should appear in *Display...* menu.

Manipulating objects

Introduction

Manipulating objects depends on whether they are based on the Archetypes subsystem or on the Dexterity subsystem.

For more information, consult the manual of the relevant subsystem:

- *Archetypes examples.*
- See Manipulating Content Objects in the Dexterity manual

Reindexing modified objects

After modifying the object, you need to reindex it in the `portal_catalog` to update the search and listing information.

Cataloging has a quirk regarding the modified metadata: when calling `reindexObject` on an object, the value for `modified` in `portal_catalog` will be set to the time of the reindex, regardless of the value of the `modified` property of the object.

In order to store the correct value you can do an extra reindex of the object with the `modified` index as parameter.

First do a normal `reindexObject`, then call it with the `modified` index explicitly:

```
object.reindexObject()  
object.reindexObject(idxs=['modified'])
```

For more information, see *** How to update this document.*

Deleting

Description

Deleting content items in Plone programmatically. How link integrity checks work and how (and when!) to avoid them.

Introduction

This document explains how to programmatically delete objects in Plone.

Deleting content by id

Deleting content objects is done by `IObjectManager`.

`IObjectManager` definition.

Example:

```
# manage_delObjects takes list of ids as an argument  
folder.manage_delObjects(["list", "of", "ids", "to", "delete"])
```

Or:

```
parent = context.aq_parent  
parent.manage_delObjects([context.getId()])
```

Permissions

The user must have *Zope 2 Delete objects* permission on the *content item* being deleted. This is checked in `Products.CMFPlone.PloneFolder.manage_delObjects()`.

Otherwise an `Unauthorized` exception is raised.

Example how to check for this permission:

```
from Products.CMFCore import permissions

hospital = self.portal.country.hospital
item = hospital.patient1

mt = getToolByName(self.portal, 'portal_membership')
if mt.checkPermission(permissions.DeleteObjects, item):
    # Can delete
    raise AssertionError("Oooops. Deletion allowed")
else:
    pass
```

Bypassing permissions

This is handy if you work e.g. in a *debug shell* and you are deleting badly behaved objects:

```
from AccessControl.SecurityManagement import newSecurityManager
admin = app.acl_users.getUserById("admin")
app.folder_sits.sitsngta.manage_delObjects("examples")
# Try harder:
# app.folder_sits.sitsngta._delObject("examples", suppress_events=True)
import transaction ; transaction.commit()
```

Bypassing link integrity check

Plone 5 handles deleting an item in its user interface only; there's no longer need to bypass link integrity programmatically.

For more information check the documentation of `plone.app.linkintegrity`.

Deleting all content in a folder

This can be a bit tricky. An example:

```
ids = folder.objectIds() # Plone 3 or older
ids = folder.keys()      # Plone 4 or newer

if len(ids) > 0:
    # manage_delObject will mutate the list
    # so we cannot give it tuple returned by objectIds()
    ids = list(ids)
    folder.manage_delObjects(ids)
```

Fail safe deleting

Sometimes deletion might fail because it dispatches events which might raise exception due to bad broken objects or badly behaving code.

`OFS.ObjectManager`, the base class for Zope folders, provides an internal method to delete objects from a folder without firing any events:

```
# Delete object with id "broken-folder" without firing any delete events
site._delObject("broken-folder", suppress_events=True)
```

The best way to clean up bad objects on your site is via a *command line script*, in which case remember to commit the transaction after removing the broken objects.

Purging old content from site

This Management Interface script allows you to find content items of certain type and delete them if they are created too long ago:

```
# Delete FeedfeederItem content items which are more than three months old

from StringIO import StringIO
import DateTime

buf = StringIO()

# DateTime deltas are days as floating points
end = DateTime.DateTime() - 30*3
start = DateTime.DateTime(2000, 1,1)

date_range_query = { 'query':(start,end), 'range': 'min:max' }

items = context.portal_catalog.queryCatalog({
    "portal_type":"FeedFeederItem",
    "created" : date_range_query,
    "sort_on" : "created" })

items = list(items)

print >> buf, "Found %d items to be purged" % len(items)

count = 0
for b in items:
    count += 1
    obj = b.getObject()
    print >> buf, "Deleting:" + obj.absolute_url() + " " + str(obj.created())
    obj.aq_parent.manage_delObjects([obj.getId()])

return buf.getvalue()
```

Below is an advanced version for old item-date-based deletion code which is suitable for huge sites. This snippet is from the `Products.feedfeeder` package. It will look for `Feedfeeder` items (automatically generated from RSS) which are older than X days and delete them.

It's based on Zope 3 page registration (sidenote: I noticed that views do not need to be based on `BrowserView` page class).

- Transaction thresholds make sure the code runs faster and does not run out of RAM
- Logging to Plone event log files
- Number of days to look into past is not hardcoded
- Manage rights needed to execute the code

You can call this view like:

```
http://localhost:9999/plonecommunity/@@feed-mega-cleanup?days=90
```

Here is the view Python source code:

```
import logging

import transaction
from zope import interface
from zope import component
import DateTime
import zExceptions

logger = logging.getLogger("feedfeeder")

class MegaClean(object):
    """ Clean-up old feed items by deleting them on the site.

    This is intended to be called from cron weekly.
    """

    def __init__(self, context, request):
        self.context = context
        self.request = request

    def clean(self, days, transaction_threshold=100):
        """ Perform the clean-up by looking old objects and deleting them.

        Commit ZODB transaction for every N objects to that commit buffer does not_
        ↪ grow
        too long (timewise, memory wise).

        @param days: if item has been created before than this many days ago it is_
        ↪ deleted

        @param transaction_threshold: How often we commit - for every nth item
        """

        logger.info("Beginning feed clean up process")

        context = self.context.aq_inner
        count = 0

        # DateTime deltas are days as floating points
        end = DateTime.DateTime() - days
        start = DateTime.DateTime(2000, 1, 1)

        date_range_query = {'query': (start, end), 'range': 'min:max'}

        items = context.portal_catalog.queryCatalog({
```



```

        "portal_type": "FeedFeederItem",
        "created": date_range_query,
        "sort_on": "created" })

items = list(items)

logger.info("Found %d items to be purged" % len(items))

for b in items:
    count += 1
    obj = b.getObject()
    logger.info("Deleting:" + obj.absolute_url() + " " + str(obj.created()))
    obj.aq_parent.manage_delObjects([obj.getId()])

    if count % transaction_threshold == 0:
        # Prevent transaction becoming too large (memory buffer)
        # by committing now and then
        logger.info("Committing transaction")
        transaction.commit()

msg = "Total %d items removed" % count
logger.info(msg)

return msg

def __call__(self):

    days = self.request.form.get("days", None)
    if not days:
        raise zExceptions.InternalError("Bad input. Please give days=60 as HTTP_
↪GET query parameter")

    days = int(days)

    return self.clean(days)

```

Then we have the view ZCML registration:

```

<page
    name="feed-mega-cleanup"
    for="Products.CMFCore.interfaces.ISiteRoot"
    permission="cmf.ManagePortal"
    class=".feed.MegaClean"
/>

```

Renaming content

Description

How to programmatically rename Plone content items

Introduction

This page tells how to rename Plone content objects and change their ids.

- This only concerns URL path ids
- Archetypes' Unique ID (UID) is not affected by the rename operation
- Title can be changed using `setTitle()` (Archetypes) or related mutator

Renaming objects

OFS interface has facilities to rename objects

- <http://svn.zope.org/Zope/trunk/src/OFS/interfaces.py?rev=105745&view=auto>
- `manage_renameObject(olddid, newid)` for one item
- `manage_renameObject([olddid, oldid2], [newid, newid2])` for rename many items
- `Products.CMFPlone.PloneFolder` overrides `manage_renameObject()` to have hooks to reindex the new object path

Warning: Security warning: “Copy or Move” permission is needed on the object by the logged in user.

Warning: New id must be a 8-bit string, not unicode. The system might accept values in invalid format.

Example how to rename object *lc* to have *-old* suffix:

```
id = lc.getId()
if not lc.cb
parent = lc.aq_parent
parent.manage_renameObject(id, id + "-old")
```

These checks performed before rename by the `manage_renameObject()`:

```
if not lc.cb_userHasCopyOrMovePermission():
    print "Does not have needed permission"
    return

if not lc.cb_isMoveable():
    # This makes sanity checks whether the object is
    # properly connected to the database
    print "Object problem"
    return
```

Warning: Testing warning: Rename mechanism relies of Persistent attribute called `_p_jar` to be present on the content object. By default, this is not the case on unit tests. You need to call `transaction.savepoint()` to make `_p_jar` appear on persistent objects.

If you don't do this, you'll receive a “CopyError” when calling `manage_renameObjects` that the operation is not supported.

Unit testing example:

```
import transaction

self.portal.invokeFactory("Document", doc)
doc = self.portal.doc

# Make sure all persistent objects have _p_jar attribute
transaction.savepoint(optimistic=True)

# Call manage_renameCode()...
```

Content types

Description

Plone's content type subsystems and creating new content types programmatically.

Introduction

Plone has two kind of content types subsystems:

- *Archetypes*-based
- *Dexterity*-based (new)

See also Plomino (later in this document).

Flexible architecture allows other kinds of content type subsystems as well.

Type information registry

Plone maintains available content types in the `portal_types` tool.

`portal_types` is a folderish object which stores type information as child objects, keyed by the `portal_type` property of the types.

`portal_factory` is a tool responsible for creating the persistent object representing the content.

[TypesTool source code](#).

Listing available content types

Often you need to ask the user to choose specific Plone content types.

Plone offers two Zope 3 vocabularies for this purpose:

`plone.app.vocabularies.PortalTypes` a list of types installed in `portal_types`

`plone.app.vocabularies.ReallyUserFriendlyTypes` a list of those types that are likely to mean something to users.

If you need to build a vocabulary of user-selectable content types in Python instead, here's how:

```
from Acquisition import aq_inner
from zope.app.component.hooks import getSite
from zope.schema.vocabulary import SimpleVocabulary, SimpleTerm
from Products.CMFCore.utils import getToolByName

def friendly_types(site):
    """ List user-selectable content types.

    We cannot use the method provided by the IPortalState utility view,
    because the vocabulary factory must be available in contexts where
    there is no HTTP request (e.g. when installing add-on product).

    This code is copied from
    https://github.com/plone/plone.app.layout/blob/master/plone/app/layout/globals/
    ↪portal.py

    @return: Generator for (id, type_info title) tuples
    """

    context = aq_inner(site)
    site_properties = getToolByName(context, "portal_properties").site_properties
    not_searched = site_properties.getProperty('types_not_searched', [])

    portal_types = getToolByName(context, "portal_types")
    types = portal_types.listContentTypes()

    # Get list of content type ids which are not filtered out
    prepared_types = [t for t in types if t not in not_searched]

    # Return (id, title) pairs
    return [ (id, portal_types[id].title) for id in prepared_types ]
```

Creating a new content type

These instructions apply to *Archetypes*-based content types.

Install ZopeSkel

Add ZopeSkel to your buildout.cfg and run buildout:

```
[buildout]
...
parts =
    instance
    zopeskel

...
[zopeskel]
recipe = zc.recipe.egg
eggs =
    PasteScript
    ZopeSkel
```

Create an archetypes product

Run the following command and answer the questions e.g. for the project name use my.product:

```
./bin/paster create -t archetype
```

Install the product

Adjust your buildout.cfg and run buildout again:

```
[buildout]
develop = my.product
...
parts =
    instance
    zopeskel
...
[instance]
eggs = my.product
```

Note: You need to install your new product using buildout before you can add a new content type in the next step. Otherwise paster complains with the following message: “Command ‘addcontent’ not known”.

Create a new content type

Deprecated since version may_2015: Use *bobtemplates.plone* instead

Change into the directory of the new product and then use paster to add a new content type:

```
cd my.product
../bin/paster addcontent contenttype
```

Related how-tos:

- <http://lionfacelemonface.wordpress.com/tutorials/zopeskel-archetypes-howto/>
- http://docs.openia.com/howtos/development/plone/creating-a-site-archetypes-object-and-contenttypes-with-paster?set_language=fi&cl=fi
- <http://www.unc.edu/~jj/plone/>

Note: Creating types by hand is not worth the trouble. Please use a code generator to create the skeleton for your new content type.

Warning: The content type name must not contain spaces. Neither the content type name or the description may contain non-ASCII letters. If you need to change these please create a translation catalog which will translate the text to one with spaces or international letters.

Debugging new content type problems

Creating types by hand is not worth the trouble.

- [Why doesn't my custom content type show up in add menu checklist.](#)

Creating new content types through-the-web

There are solutions for non-programmers and Plone novices to create their content types.

Dexterity

- <http://docs.plone.org/external/plone.app.dexterity/docs/>
- Core feature
- Use Dexterity control panel in site setup
- Use bobtemplates.plone

Plomino (Archetypes-based add-on)

- With Plomino you can make an entire web application that can organize & manipulate data with very limited programming experience.
- <http://www.plomino.net/>
- http://www.youtube.com/view_play_list?p=469DE37C742F31D1

Implicitly allowed

Implicitly allowed is a flag specifying whether the content is globally addable or must be specifically enabled for certain folders.

The following example allows creation of *Large Plone Folder* anywhere at the site (it is disabled by default). For available properties, see `TypesTool._advanced_properties`.

Example:

```
portal_types = self.context.portal_types
lpf = portal_types["Large Plone Folder"]
lpf.global_allow = True # This is "Globally allowed" property
```

Constraining the addable types per type instance

For the instances of some content types, the user may manually restrict which kinds of objects may be added inside. This is done by clicking the *Add new...* link on the green edit bar and then choosing *Restrictions...*

This can also be done programmatically on an instance of a content type that supports it.

First, we need to know whether the instance supports this.

Example:

```
from Products.Archetypes.utils import shasattr # To avoid acquisition
if shasattr(context, 'canSetConstrainTypes'):
    # constrain the types
    context.setConstrainTypesMode(1)
    context.setLocallyAllowedTypes(('News Item',))
```

If `setConstrainTypesMode` is 1, then only the types enabled by using `setLocallyAllowedTypes` will be allowed.

The types specified by `setLocallyAllowedTypes` must be a subset of the allowable types specified in the content type's FTI (Factory Type Information) in the `portal_types` tool.

If you want the types to appear in the `:guilabel: Add new. .` dropdown menu, then you must also set the immediately addable types. Otherwise, they will appear under the *more* submenu of *Add new...*

Example:

```
context.setImmediatelyAddableTypes(('News Item',))
```

The immediately addable types must be a subset of the locally allowed types.

To retrieve information on the constrained types, you can just use the accessor equivalents of the above methods.

Example:

```
context.getConstrainTypesMode()
context.getLocallyAllowedTypes()
context.getImmediatelyAddableTypes()
context.getDefaultAddableTypes()
context.allowedContentTypes()
```

Be careful of Acquisition. You might be acquiring these methods from the current instance's parent. It would be wise to first check whether the current object has this attribute, either by using `shasattr` or by using `hasattr` on the object's base (access the base object using `aq_base`).

The default addable types are the types that are addable when `constrainTypesMode` is 0 (i.e not enabled).

For more information, see **Products/CMFPlone/interfaces/constraints.py**

Workflows

Description

Programming workflows in Plone.

Introduction

The `DCWorkflow` product manages the default Plone workflow system.

A workflow state is not directly stored on the object. Instead, a separate `portal_workflow` tool must be used to access a workflow state. Workflow look-ups involve an extra database fetch.

For more information, see

- <http://www.martinaspeli.net/articles/dcworkflows-hidden-gems>

Creating workflows

The recommended method is to use `portal_workflow` in the Management Interface to construct the workflow through the web and then you can export it using GenericSetup's `portal_setup` tool.

Include necessary parts from exported `workflows.xml` and `workflows` folder in your add-on product GenericSetup profile (add-on folder `profiles/default`).

Model the workflow online

Go to `'http:yourhost.com:8080/yourPloneSiteName/portal_workflow/manage_main'`, copy and paste `'simple_publication_workflow'`, to have a skeleton for start-off, rename `'copy_of_simple_publication_workflow'` to `'your_workflow'` or add a new workflow via the dropdown-menu and have a tabula rasa.

Add and remove states and transitions, assign permissions etc.

Putting it in your product

Go to `'http:yourhost.com:8080/yourPloneSiteName/portal_setup/manage_exportSteps'`, check `'Workflow Tool'` and hit `'Export selected steps'`, unzip the downloaded file and put the `definition.xml`-file in `'your/product/profiles/default/workflows/your_workflow/'` (you'll need to create the latter two directories).

Configure workflow via GenericSetup

Assign a workflow

In `your/product/profiles/default/workflows.xml`, insert:

```
<?xml version="1.0" ?>
<object name="portal_workflow" meta_type="Plone Workflow Tool" purge="False">

    <object name="your_workflow" meta_type="Workflow" />

</object>
```

Assigning a workflow globally as default

In `your/product/profiles/default/workflows.xml`, add:

```
<object name="portal_workflow">
    (...)
    <bindings>
        <default>
            <bound-workflow workflow_id="simple_publication_workflow" />
        </default>
    </bindings>
</object>
```


Binding a workflow to a content type

Example with GenericSetup *workflows.xml*

```
<?xml version="1.0"?>
<object name="portal_workflow" meta_type="Plone Workflow Tool">
  <bindings>
    <type type_id="Image">
      <bound-workflow workflow_id="plone_workflow" />
    </type>
  </bindings>
</object>
```

Disabling workflow for a content type

If a content type doesn't have a workflow it uses its parent container security settings. By default, content types Image and File have no workflow.

Workflows can be disabled by leaving the workflow setting empty in portal_workflow in the Management Interface.

Example how to do it with GenericSetup *workflows.xml*

```
<?xml version="1.0"?>
<object name="portal_workflow" meta_type="Plone Workflow Tool">
  <property
    name="title">Contains workflow definitions for your portal</property>
  <bindings>
    <!-- Bind nothing for these content types -->
    <type type_id="Image"/>
    <type type_id="File"/>
  </bindings>
</object>
```

Updating security settings after changing workflow

Through the web this would be done by going to the Management Interface > portal_workflow > update security settings

To update security settings programmatically use the method `updateRoleMappings`. The snippet below demonstrates this:

```
from Products.CMFCore.utils import getToolByName
# Do this after installing all workflows
wf_tool = getToolByName(self, 'portal_workflow')
wf_tool.updateRoleMappings()
```

Programmatically

Getting the current workflow state

Example:

```
workflowTool = getToolByName(self.portal, "portal_workflow")
# Returns workflow state object
status = workflowTool.getStatusOf("plone_workflow", object)
# Plone workflows use variable called "review_state" to store state id
# of the object state
state = status["review_state"]
assert state == "published", "Got state:" + str(state)
```

Filtering content item list by workflow state

Here is an example how to iterate through content item list and let through only content items having certain state.

Note: Usually you don't want to do this, but use content aware folder listing method or portal_catalog query which does filtering by permission check.

Example:

```
portal_workflow = getToolByName(self.context, "portal_workflow")

# Get list of all objects
all_objects = [ obj for obj in self.all_content if ISubjectGroup.providedBy(obj) or
↳ IFeaturedCourses.providedBy(obj) == True ]

# Filter objects by workflow state (by hand)
for obj in all_objects:
    status = portal_workflow.getStatusOf("plone_workflow", obj)
    if status and status.get("review_state", None) == "published":
        yield obj
```

Changing workflow state

You cannot directly set the workflow to any state, but you must push it through legal state transitions.

Security warning: Workflows may have security assertions which are bypassed by admin user. Always test your workflow methods using a normal user.

Example how to publish content item banner:

```
from Products.CMFCore.WorkflowCore import WorkflowException

workflowTool = getToolByName(banner, "portal_workflow")
try:
    workflowTool.doActionFor(banner, "publish")
except WorkflowException:
    # a workflow exception is risen if the state transition is not available
    # (the sampleProperty content is in a workflow state which
    # does not have a "submit" transition)
    logger.info("Could not publish:" + str(banner.getId()) + " already published?")
    pass
```

Example how to submit to review:

```

from Products.CMFCore.WorkflowCore import WorkflowException

portal.invokeFactory("SampleContent", id="sampleProperty")

workflowTool = getToolByName(context, "portal_workflow")
try:
    workflowTool.doActionFor(portal.sampleProperty, "submit")
except WorkflowException:
    # a workflow exception is risen if the state transition is not available
    # (the sampleProperty content is in a workflow state which
    # does not have a "submit" transition)
    pass

```

Example how to cause specific transitions based on another event (e.g. a parent folder state change). This code must be part of your product's trusted code not a workflow script because of the permission issues mentioned above. See also see [Events](#)

```

# Subscribe to the workflow transition completed action
from five import grok
from Products.DCWorkflow.interfaces import IAfterTransitionEvent
from Products.CMFCore.interfaces import IFolderish

@grok.subscribe(IFolderish, IAfterTransitionEvent)
def make_decisions_visible(context, event):
    if (event.status['review_state'] != 'cycle_complete'):
        #nothing to do
        return
    children = context.getFolderContents()
    wftool = context.portal_workflow
    #loop through the children objects
    for obj in children:
        state = obj.review_state
        if (state=="alternate_invisible"):
            # below is workaround for using getFolderContents() which provides a
            # 'brain' rather than an python object. Inside if to avoid overhead
            # of getting object if do not need it.
            what = context[obj.id]
            wftool.doActionFor(what, 'to_alternate')
        elif (state=="denied_invisible"):
            what = context[obj.id]
            wftool.doActionFor(what, 'to_denied')
        elif (...

```

Gets the list of ids of all installed workflows

Useful to test if a particular workflow is installed:

```

# Get all site workflows
ids = workflowTool.getWorkflowIds()
self.assertIn('link_workflow', ids, "Had workflows " + str(ids))

```

Getting default workflow for a portal type

Get default workflow for the type:

```
chain = workflowTool.getChainForPortalType(ExpensiveLink.portal_type)
self.assertEqual(chain, ('link_workflow',), "Had workflow chain" + str(chain))
```

Getting workflows for an object

How to test which workflow the object has:

```
# See that we have a right workflow in place
workflowTool = getToolByName(context, "portal_workflow")
# Returns tuple of all workflows assigned for a context object
chain = workflowTool.getChainFor(context)

# there must be only one workflow for our object
self.assertEqual(len(chain), 1)

# this must must be the workflow name
self.assertEqual(chain[0], 'link_workflow', "Had workflow " + str(chain[0]))
```

Via HTTP

Plone provides a `workflow_action` script which is able to trigger the status modification through an HTTP request (browser address bar).

Example:

```
http://localhost:9020/site/page/content_status_modify?workflow_action=publish
```

Content Identification (ids)

Description

Different ids, UUIDs, integer ids or whatever can identify your Plone content and give access to it.

Introduction

Id

Content id generally refers the item id **within the folder**.

Together with folder path this identifies the content in unique way.

Naturally, this id changes when the content is renamed or moved.

Use *traversing* to resolve object by path+id.

UUID and UUID

UID is a unique, non-human-readable identifier for a content object which stays on the object even if the object is moved.

Plone uses UUIDs for

- Storing content-to-content references (Archetypes, ReferenceField)
- Linking by UUIDs - this enables persistent links even though the object is moved
- Plain UUID is supported by Archetypes only and is based on `reference_catalog`
- UUID is supported by Archetypes and Dexterity both and you should use this for new projects

UUIDs are available for Archetypes content and unified UUIDs for both Archetypes and Dexterity content items since `plone.app.dexterity` version 1.1.

Note: If you have pre-Dexterity 1.1 content items you must run a migration step in `portal_setup` to give them UUIDs.

To get object UUID you can use `plone.app.uuid` package.

Getting object UUID:

```
from plone.uuid.interfaces import IUUID

# BrowserView helper method
def getUID(self):
    """ AT and Dexterity compatible way to extract UUID from a content item """
    # Make sure we don't get UID from parent folder accidentally
    context = self.context.aq_base
    # Returns UID of the context or None if not available
    # Note that UID is always available for all Dexterity 1.1+
    # content and this only can fail if the content is old not migrated
    uuid = IUUID(context, None)
    return uuid
```

Looking up object by UUID:

Use `plone.app.uuid.utils.uuidToObject`:

```
from plone.app.uuid.utils import uuidToObject

...
obj = uuidToObject(uuid)
if not obj:
    # Could not find object
    raise RuntimeError(u"Could not look-up UUID:", uuid)
```

More info:

- <http://stackoverflow.com/questions/8618917/portal-catalog-unique-ids-for-both-archetypes-and-dexterity-content>

UUID Acquisition Problem With Dexterity Content Types

Make sure your Dexterity content type has the `plone.app.referenceablebehavior.interfaces.IReferenceable` behavior enabled.

If not, when querying for an object's UUID, you will get its parent UUID.

Then you can end up with a lot of objects with the same UUID as their parent.

If you run into this issue, here's an easy upgrade step to fix it:

```
import transaction
from plone.uuid.handlers import addAttributeUUID
from Products.CMFCore.utils import getToolByName

...
def recalculate_uuids(setup_tool):

    # Re-import types definition, so IReferenceable is enabled.
    setup_tool.runImportStepFromProfile(
        "profile-my.package:default",
        'typeinfo')

    catalog = getToolByName(setup_tool, 'portal_catalog')
    for index, brain in enumerate(catalog(portal_type="my.custom.content.type")):
        obj = brain.getObject()

        if not getattr(obj, '_plone.uuid', None) is None:
            # If an UUID has already been calculated for this object, remove it
            delattr(obj, '_plone.uuid')

        # Recalculate object's UUID
        addAttributeUUID(obj, None)
        obj.reindexObject(idxs=['UUID'])

        if index % 100 == 0:
            # Commit every 100 items
            transaction.commit()

    # Commit at the end
    transaction.commit()
```

Make sure to have the IReferenceable behavior listed in the content type XML definition before running the upgrade step.

Note: This upgrade step will recalculate the UUID for all “my.custom.content.type” objects.

intids

Integer ids (“intids”) are fast look-up ids provided by `plone.app.intid` and `five.intid` packages.

Instead of relying on globally unique identifier strings (UIDs) they use 64-bit integers, making low-level resolution faster.

- <https://github.com/plone/plone.app.intid>
- <http://stackoverflow.com/questions/8629390/how-to-use-intids>

Ownership of content

Description

Programmatically manipulate Plone content item’s ownership

Introduction

Each content item has an owner user.

Owned item instances are of subclass of `AccessControl.Owned`

- <http://svn.zope.org/Zope/trunk/src/AccessControl/Owned.py?rev=96262&view=auto>

Getting the owner of the item

Example:

```
# Returns PropertyUser for Zope admin
# Returns PloneUser for normal users object
context.getOwner()
```

Changing ownership of content

You can use `AccessControl.Owner.changeOwnership`:

```
changeOwnership(self, user, recursive=0)
```

User is `PropertyUser` object.

Example:

```
# Get the user handle from member data object
user = member.getUser()

# Make the member owner of his home folder
home_folder.changeOwnership(user, recursive=False)
home_folder.reindexObjectSecurity()
```

Warning: This only changes the owner attribute, not the role assignments. You need to change those too.

Example how to add ownership for additional user using local roles:

```
home_folder.manage_setLocalRoles(username, ["Owner",])
home_folder.reindexObjectSecurity()
```

Note: This does not update Dublin Core metadata fields like creator.

Contributors

Contributors is an automatically managed list where persons, who have been editing in the past, real names are listed. Contributors data is available as Python list of real names.

Note: Contributors does not store user references, because one might want to maintain contributor data even after the user has been deleted.

Some sample code:

```
def format_contributors(contribs):
    """
    @return: String of comma separated list of all contributors
    """

    if len(contribs) == 0:
        return None

    return ", ".join(contribs)

data = {
    "contributors" : format_contributors(obj.Contributors()),
}
```

```
<span tal:condition="o/contributors">
    <span tal:replace="o/contributors">Jim Smith, Jane Doe</span>
</span>
```

Timestamps

Description

How to read created and modified timestamps on Plone content items programmatically

local

- *Timestamps*
 - *Introduction*
 - *Last modification date*
 - * *Setting modification date explicitly*
 - * *Viewlet example*
 - *Creation date*
 - *IsExpired()*

Introduction

Here are some useful timestamps you can extract from content objects and examples how to use them.

Timestamps are part of metadata. For Archetypes, metadata is defined in [ExtensibleMetadata](#).

Zope 2 DateTime date objects are used.

Last modification date

`Products.Archetypes.ExtensibleMetadata.modified()` function will give the last modification date as Zope `DateTime` object. This is part of Dublin Core metadata.

Example (Zope console debug mode):

```
>>> app.yoursite.yourpage.modified()
DateTime('2009/02/04 10:56:25.740 Universal')
```

Setting modification date explicitly

You might want to manual set modification date

- When you migrate content
- When you edit content subobjects and want to update the timestamp of parent object to reflect this changes

Example (Zope console debug mode, assume `obj` is Archetypes content item):

```
>>> obj.modified()
>>> DateTime('2009/10/05 16:18:32.813 GMT+2')

>>> import DateTime

>>> now = DateTime.DateTime()
>>> now
>>> DateTime('2010/01/20 12:58:38.033 GMT+2')

>>> obj.setModificationDate(now)
>>> obj.modified()
>>> DateTime('2010/01/20 12:58:38.033 GMT+2')
```

Viewlet example

Below is an example how to create a custom last modified viewlet.

Viewlet code:

```
from zope.component import getMultiAdapter
from plone.app.layout.viewlets.common import ViewletBase

class LastModifiedViewlet(ViewletBase):
    """ Viewlet to change the document last modification time.
    """

    def modified(self):
        """
        https://github.com/plone/Products.CMFPlone/blob/master/Products/CMFPlone/
↪browser/ploneview.py

        @return: Last modified as a string, local time format
        """

        # Get Plone helper view
```

```
# which we use to convert the date to local format
plone = getMultiAdapter((self.context, self.request), name="plone")

time = self.context.modified()

return plone.toLocalizedTime(time)
```

Template (lastmodified.py):

```
<div id="last-modified">
    Last modified: <span tal:content="view/modified" />
</div>
```

Viewlet registration:

```
<!-- Last modification date, register only for contentish context objects -->
<browser:viewlet
    name="yourapp.lastmodified"
    for="Products.CMFCore.interfaces.IContentish"
    manager="plone.app.layout.viewlets.interfaces.IBelowContent"
    template="viewlets/lastmodified.pt"
    class=".common.LastModifiedViewlet"
    permission="zope2.View"
/>
```

CSS:

```
#last-modified {
    text-align: right;
    font-size: 80%;
    color: #888;
}
```

Creation date

`Products.Archetypes.ExtensibleMetadata.created()` function will give the creation date as Zope `DateTime` object. This is part of Dublin Core metadata.

Example (Zope console debug mode):

```
>>> app.yoursite.yourpage.created()
DateTime('2009/02/04 10:56:25.740 Universal')
```

IsExpired()

- <https://github.com/plone/Products.CMFPlone/blob/master/Products/CMFPlone/utils.py#L112>

Dynamic views

Description

How to programmatically change the active view of a Plone content item

Introduction

Dynamic views are views which the content editor can choose for his or her content from the *Display...* drop-down menu in the green edit frame.

By default, Plone comes with dynamic views for:

- Folder listing
- Summary
- Photo album
- etc.

The default view can be also a content item picked from the folder.

Available content item types can be managed from the: Site Setup Control Panel -> Content Rules (site.com/@@content-controlpanel) -> Select your new type from the drop down menu -> Click the “Can be used as default page” checkbox.

Permission for changing the view template of an item

A user needs the *Modify view template* permission to use the dynamic view dropdown. If you want to restrict this ability, grant or revoke this permission as appropriate.

This can be useful for some content types like Dexterity ones, where dynamic views are enabled by default, and the easiest way to disable them is using this permission.

Default dynamic views

Plone supports a few dynamic views for folders out of the box:

- Summary view (`folder_summary_view`)
- Tabular view (`folder_tabular_view`)
- Album view (`atct_album_view`)
- Listing (`folder_listing`)
- Full view (`folder_full_view`)

These are defined in [portal_types information](#) for the *Folder* content type and mapped to the *Display* menu all over in ZCML using `browser:menuItem` as described below.

Newly created folders have this dynamic view applied:

- `Products.CMFPlone/skins/plone_content/folder_summary_view.pt` (a non-view based old style Zope 2 page template)

More info

- [Overriding views](#)

Creating a dynamic view

Here are instructions how to create your own dynamic view.

There is also an example product [Listless view](#), which provides “no content listing” view for Folder content types.

Registering a dynamic view menu item

In order to be able to register dynamic views, your content type must support them.

To do this, the content type should subclass `Products.CMFDynamicViewFTI.browserdefault.BrowserDefaultMixin`.

Then, you need to register a dynamic view menu item with the corresponding view in your `configure.zcml`:

```
<browser:menuItem
    for="Products.ATContentTypes.interface.IATFolder"
    menu="plone_displayviews"
    title="Product listing"
    action="@@product_listing"
    description="List folder contents as product summary view"
/>
```

Note: `Products.ATContentTypes` uses a non-standard name for the interfaces package. There, it is interface, while all other packages use interfaces.

The view must be listed in `portal_types` for the content type. In this case, we should enable it for Archetypes folders using the following `GenericSetup XML`: `profiles/default/types/Folder.xml`.

Note that you don't need to copy the whole `Folder.xml` / `Topic.xml` from `Products/CMFPlone/profiles/default/types`. Including the changed `view_methods` in the XML code is enough.

You can also change this through `portal_types` in the Management Interface.

Note: `view_methods` must not have the `@@view` signature in their method name.

```
<?xml version="1.0"?>
<object name="Folder"
  xmlns:i18n="http://xml.zope.org/namespaces/i18n"
  i18n:domain="plone"
  meta_type="Factory-based Type Information with dynamic views" >
  <property name="view_methods" purge="False">
    <!-- We retrofit these new views for Folders in portal_types info -->
    <element value="product_listing"/>
  </property>
</object>
```

Also, if you want *Collections* to have this listing, you need to add the following `profiles/default/types/Topic.xml`.

```
<?xml version="1.0"?>
<object name="Topic"
  xmlns:i18n="http://xml.zope.org/namespaces/i18n"
  i18n:domain="plone"
  meta_type="Factory-based Type Information with dynamic views" >
  <property name="view_methods">
    <element value="folder_listing"/>
    <element value="folder_summary_view"/>
    <element value="folder_tabular_view"/>
    <element value="atct_album_view"/>
    <element value="atct_topic_view"/>
  </property>
</object>
```

```

    <!-- We retrofit these new views for Folders in portal_types info -->
    <element value="product_listing"/>

    </property>
</object>

```

Working around broken default view

If you manage to:

- Create a new view
- set it to the default as a folder
- and this view has a bug

... you cannot access the folder anymore, because you are taken to the broken view stack trace instead instead of rendering the green edit menubar.

The fix is to reset the view by browsing to the `select_default_view` directly. Access your folder like this:

```
http://servername/plonesite/folder/select_default_view
```

Checking that your view is available

`Products.CMFDynamicViewFTI.browserdefault.BrowserDefaultMixin.getAvailableLayouts()` returns the list of known layouts in the following format:

```

[('folder_summary_view', 'Summary view'),
 ('folder_tabular_view', 'Tabular view'),
 ('atct_album_view', 'Thumbnail view'),
 ('folder_listing', 'Standard view'),
 ('product_listing', u'Product listing')]

```

To see if your view is available, check it against the ids from that result:

```

layout_ids = [id for id, title in self.portal.folder.getAvailableLayouts() ]
self.assertTrue("product_list" in layout_ids)

```

Getting active layout

```

>>> self.portal.folder.getLayout()
'atct_album_view'

```

Changing default view programmatically

```
self.portal.folder.setLayout("product_listing")
```

Default page

The default page is a *content item* chosen to be displayed when the visitor arrives at a URL without any subpages or views selected.

This is useful if you are doing the folder listing manually and you want to replace the default view.

The `default_page` helper view can be used to manipulate default pages.

Getting the default page:

```
# Filter out default content
container = self.getListingContainer()
default_page_helper = getMultiAdapter(
    (container, self.request), name='default_page')

# Return content object which is the default page or None if not set
default_page = default_page_helper.getDefaultPage(container)
```

Another example how to use this:

```
from Products.CMFCore.interfaces import IFolderish

def hasTabs(self):
    """Determine whether the page itself, or default page, in the case
    of folders, has setting showTabs set true.

    Show tab setting defined in dynamicpage.py.
    """

    page = self.context

    try:
        if IFolderish.providedBy(self.context):
            folder = self.context
            default_page_helper = getMultiAdapter(
                (folder, self.request), name='default_page')
            page_name = default_page_helper.getDefaultPage(folder)
            page = folder[page_name]
    except:
        pass

    tabs = getattr(page, "showTabs", False)

    return tabs
```

Setting the default page can be done by calling the `setDefaultPage` on the folder, passing id of the default page:

```
folder.setDefaultPage("my_content_id")
```

More information can be found in

- <https://github.com/plone/plone.app.layout/blob/master/plone/app/layout/globals/context.py>
- <https://github.com/plone/plone.app.layout/blob/master/plone/app/layout/navigation/defaultpage.py>

Disabling dynamic views

Add to your content type class:

```
def canSetDefaultPage(self):
    """
    Override BrowserDefaultMixin because default page stuff doesn't make
    sense for topics.
    """
    return False
```

Setting a view using marker interfaces

If you need to have a view for few individual content items only, it is best to do this using marker interfaces.

Create a marker interface in python:

```
from zope.interface import Interface

class IMyMarkerInterface(Interface):
    """Used to create a specific view for a generic content type"""
```

Register the marker interface with ZCML, see [marker interfaces](#):

```
<interface interface="my.package.interfaces.IMyMarkerInterface" />
```

Register the view against a marker interface:

```
<browser:page
    class="my.package.browser.views.MySpecificView"
    for="my.package.interfaces.IMyMarkerInterface"
    layer="my.package.interfaces.IBrowserLayer"
    name="my-custom-view"
    permission="zope2.View"
    template="view.pt"
/>
```

- Assign this marker interface to a content item using the Management Interface, via the Interfaces tab or with Python code:

```
from my.package.interfaces import IMyMarkerInterface
from plone import api
from Products.Five.utilities.interfaces import IMarkerInterfaces

portal = api.portal.get()
folder = portal['my-folder']
adapter = IMarkerInterfaces(folder)
adapter.update(add=(IMyMarkerInterface, ))
```

- If the view should be the default view for that given object, add a `layout` property with value `my-custom-view`. To do the same with python, see [Changing default view programmatically](#).

Migration script from default view to another

Below is a script snippet which allows you to change the default view for all folders to another type. You can execute the script through the Management Interface as a Python script.

Script code:

```
from StringIO import StringIO

original = 'fancy_zoom_view'
target = 'atct_album_view'
for brain in context.portal_catalog(portal_type="Folder"):
    obj = brain.getObject()
    if getattr(obj, "layout", None) == original:
        print "Updated:" + obj.absolute_url()
        obj.setLayout(target)
return printed
```

This will allow you to migrate from `collective.fancyzoom` to Plone 4's default album view or `Products.PipBox`.

Method aliases

Method aliases allow you to redirect basic actions (view, edit) to content type specific views. Aliases are configured in `portal_types`.

Other resources

- <https://plone.org/documentation/how-to/how-to-create-and-set-a-custom-homepage-template-using-generic-setup>
- CMFDynamicView [plone.org product page](#)

Behaviors

Behaviors allow you to extend the functionality of existing content.

- [Tutorial](#)
- [Source code](#)
- [Good known component version set for plone.behavior](#)

History and versioning

Introduction

Plone versioning allows you to go back between older edits of the same content object.

Versioning allows you to restore and diff previous copies of the same content. [More about CMFEditions here.](#)

See also

- [Versioning tutorial for custom content types.](#)

Checking whether versioning is enabled

The following check is performed by `update_versioning_before_edit` and `update_versioning_on_edit` scripts:


```
pr = context.portal_repository

isVersionable = pr.isVersionable(context)

if pr.supportsPolicy(context, 'at_edit_autoversion') and isVersionable:
    # Versioning should work
    pass
else:
    # Something is wrong...
    pass
```

Inspecting versioning policies

Example:

```
portal_repository = context.portal_repository
map = portal_repository.getPolicyMap()
for i in map.items(): print i
```

Will output (inc. some custom content types):

```
('File Disease Description', ['at_edit_autoversion', 'version_on_revert'])
('Document', ['at_edit_autoversion', 'version_on_revert'])
('Free Text Disease Description', ['at_edit_autoversion', 'version_on_revert'])
('ATDocument', ['at_edit_autoversion', 'version_on_revert'])
('Diagnose Description', ['at_edit_autoversion', 'version_on_revert'])
('ATNewsItem', ['at_edit_autoversion', 'version_on_revert'])
('Link', ['at_edit_autoversion', 'version_on_revert'])
('News Item', ['at_edit_autoversion', 'version_on_revert'])
('Event', ['at_edit_autoversion', 'version_on_revert'])
```

How versioning (CMFEditions) works

- <http://svn.zope.de/plone.org/collective/Products.CMFEditions/trunk/doc/DevelDoc.html>

Note: You might actually want to check out the package to get your web browser to properly read the file.

Getting the complete revision history for an object

You may find yourself needing to (programmatically) get some/all of a content object's revision history. The content history view can be utilised to do this; this view is the same one that is visible through Plone's web interface at @@contenthistory (or indirectly on @@historyview). This code works with Plone 4.1 and has been utilised for exporting raw content modification information:

```
from plone.app.layout.viewlets.content import ContentHistoryView
context = portal['front-page']
print ContentHistoryView(context, context.REQUEST).fullHistory()
```

If you want to run this from somewhere without a REQUEST available, such as the *Plone/Zope debug console*, then you'll need to fake a request and access level accordingly. Note the subtle change to using `ContentHistoryViewlet` rather than `ContentHistoryView` - we need to avoid initialising an entire view

because this involves component lookups (and thus, pain). We also need to fake our security as well to avoid anything being left out from the history.

```
from plone.app.layout.viewlets.content import ContentHistoryViewlet
from zope.publisher.browser import TestRequest
from AccessControl.SecurityManagement import newSecurityManager

admin = app.acl_users.getUser('webmaster')
request = TestRequest()
newSecurityManager(request, admin)

portal = app.ands
context = portal['front-page']
chv = ContentHistoryViewlet(context, request, None, None)
#These attributes are needed, the fullHistory() call fails otherwise
chv.navigation_root_url = chv.site_url = 'http://www.foo.com'
print chv.fullHistory()
```

The end result should look something like this, which has plenty of tasty morsels to pull apart and use:

```
[{'action': u'Edited',
  'actor': {'description': '',
            'fullname': 'admin',
            'has_email': False,
            'home_page': '',
            'language': '',
            'location': '',
            'username': 'admin'},
  'actor_home': 'http://www.foo.com/author/admin',
  'actorid': 'admin',
  'comments': u'Initial revision',
  'diff_current_url': 'http://foo/Plone5/front-page/@@history?one=current&two=0',
  'preview_url': 'http://foo/Plone5/front-page/versions_history_form?version_id=0',
  ↪ #version_preview',
  'revert_url': 'http://foo/Plone5/front-page/revertversion',
  'time': 1321397285.980262,
  'transition_title': u'Edited',
  'type': 'versioning',
  'version_id': 0},
 {'action': 'publish',
  'actor': {'description': '',
            'fullname': '',
            'has_email': False,
            'home_page': '',
            'language': '',
            'location': '',
            'username': 'admin'},
  'actor_home': 'http://www.foo.com/author/admin',
  'actorid': 'admin',
  'comments': '',
  'review_state': 'published',
  'state_title': 'Published',
  'time': DateTime('2011/11/15 09:49:8.023381 GMT+10'),
  'transition_title': 'Publish',
  'type': 'workflow'},
 {'action': None,
  'actor': {'description': '',
            'fullname': '',
            'has_email': False,
```

```
        'home_page': '',
        'language': '',
        'location': '',
        'username': 'admin'},
'actor_home': 'http://www.foo.com/author/admin',
'actorid': 'admin',
'comments': '',
'review_state': 'private',
'state_title': 'Private',
'time': DateTime('2011/11/15 09:49:8.005597 GMT+10'),
'transition_title': u'Create',
'type': 'workflow']}]
```

For instance, you can determine who the last person to modify this Plone content was by looking at the first list element (and get all their details from the actor information). Refer to the source of `plone.app.layout.viewlets.content` for more information about `ContentHistoryView`, `ContentHistoryViewlet` and `WorkflowHistoryViewlet`. Using these other class definitions, you can see that you can get just the workflow history using `.workflowHistory()` or just the revision history using `.revisionHistory()`.

Purging history

- <http://stackoverflow.com/questions/9683466/purging-all-old-cmfeditions-versions>

Importing and exporting content

Description

Importing and exporting content between Plone sites and other CMS systems

Introduction

Goal: you want to import and export content between Plone sites.

- If both sites have identical version and add-on product configuration you can use the Management Interface export/import
- If they don't (e.g. have different Plone version on source and target site), you need to use add-on products to export and import the content to a common format, e.g. JSON files.

Zope 2 import / export

Zope 2 can import/export parts of the site in `.zexp` format. This is basically Python pickle data of the exported objects. The data is a raw dump of Python internal data structures, which means that the source and the target Plone versions must be compatible. For example, a export from Plone 3 to Plone 4 is not possible.

To export objects from a site to another, do the following:

- In the Management Interface, navigate to the Folder, which holds the object to be exported.
- Tick the checkbox for a object to be exported.
- Click *Import / Export*

- Export as .zexp.
- Zope 2 will tell you the path where .zexp was created on the server.
- Zope .zexp to *youranothersite*/var/instance/import folder
- Go to the Management Interface root of your other site
- Press Import / Export
- In *Import from file* you should see now your .zexp file
- Import it
- Go to portal_catalog -> Advanced tab. *Clear and Rebuild* the catalog (raw Zope pickle does not know about anything living inside the catalog)

collective.transmogrifier

On it's own `collective.transmogrifier` isn't an import tool, rather a generic framework for creating pipelines to process data. Pipeline configs are .ini-style files that join together blueprints to quickly create a tool for processing data.

The following add-ons make it useful in a Plone context:

- `plone.app.transmogrifier` provides integration with GenericSetup, so you can run pipelines as part of import steps, and some useful blueprints.
- `quintagroup.transmogrifier` also provides it's own Plone integration, and some useful blueprints. See the site for some example configs for migration.
- `transmogrify.dexterity` provides some blueprints relevant to Dexterity types, and has some default pipelines for you to use.
- `collective.jsonmigrator` is particularly useful when the old site is not able to install `collective.transmogrifier`, as `collective.jsonmigrator` has a low level of dependencies for that end of the migration.

transmogrify.dexterity: CSV import

`transmogrify.dexterity` will register the pipeline `transmogrify.dexterity.csvimport`, which can be used to import from CSV to dexterity objects.

For more information on using, see [the package documentation](#).

transmogrify.dexterity: JSON import/export

`transmogrify.dexterity` also contains some `quintagroup.transmogrifier` pipeline configs. To use these, first install both `quintagroup.transmogrifier` and `transmogrify.dexterity`, then add the following to your ZCML:

```
<include package="transmogrify.dexterity.pipelines" file="files.zcml" />
```

Then the “Content (transmogrifier)” generic setup import / export will import / export site content to JSON files.

For more information on using, see [this transmogrify blog post](#).

quintagroup.transmogrifier: Exporting single folder only

Here is explained how to export and import [Plone CMS](#) folders between different Plone versions, or different CMS systems, using XML based content marshalling and [quintagroup.transmogrifier](#).

This overcomes some problems with Zope management based export/import which uses [Python pickles](#) and thus needs identical codebase on the source and target site. Exporting and importing between Plone 3 and Plone 4 is possible.

You can limit export to cover source content to with arbitrary [portal_catalog](#) conditions. If you limit source content by path you can effectively export single folder only.

The recipe described here assumes the exported and imported site have the same path for the folder. Manually rename or move the folder on source or target to change its location.

Note: The instructions here requires [quintagroup.transmogrify](#) version 0.4 or later.

Source site

Execute these actions on the source Plone site.

Install [quintagroup.transmogrifier](#) via buildout and Plone add-on control panel.

Go to *Site setup > Content migration*.

Edit export settings. Remove unnecessary pipeline entries by looking the example below. Add a new `catalogsource` blueprint. The `exclude-contained` option makes sure we do not export unnecessary items from the parent folders:

```
[transmogrifier]
pipeline =
    catalogsource
    fileexporter
    marshaller
    datacorrector
    writer
    EXPORTING

[catalogsource]
blueprint = quintagroup.transmogrifier.catalogsource
path = query= /isleofback/ohjeet
exclude-contained = true
```

Also we need to include some field-level excluding bits for the folders, because the target site does not necessary have the same content types available as the source site and this may prevent setting up folderish content settings:

```
[marshaller]
blueprint = quintagroup.transmogrifier.marshaller
exclude =
    immediatelyAddableTypes
    locallyAllowedTypes
```

You might want to remove other, unneeded blueprints from the export pipeline. For example, `portletexporter` may cause problems if the source and target site do not have the same portlet code.

Go to the *Management Interface > portal_setup > Export* tab. Check Content (transmogrifier) step. Press *Export Selected Steps* at the bottom of the page. Now a `.tar.gz` file will be downloaded.

During the export process `instance.log` file is updated with status info. You might want to follow it in real-time from UNIX command line

```
tail -f var/log/instance.log
```

In log you should see entries running like:

```
2010-12-27 12:05:30 INFO EXPORTING _path=sisalto/ohjeet/yritys/yritysten-tuotetiedot/
↪tuotekortti
2010-12-27 12:05:30 INFO EXPORTING
Pipeline processing time: 00:00:02
    94 items were generated in source sections
    94 went through full pipeline
    0 were discarded in some section
```

Target site

Execute these actions on the target Plone site.

Install `quintagroup.transmogrifier` via buildout and Plone add-on control panel.

Open target site `instance.log` file for monitoring the import process

```
tail -f var/log/instance.log
```

Go to the *Management Interface* > *portal_setup* > *Import* tab.

Choose downloaded `setup_toolxxx.tar.gz` file at the bottom of the page, for *Import uploaded tarball* input.

Run import and monitoring log file for possible errors. Note that the import completes even if the target site would not able to process incoming content. If there is a serious problem the import seems to complete successfully, but no content is created.

Note: Currently export/import is not perfect. For example, the Management Interface content type icons are currently lost in the process. It is recommended to do a test run on a staging server before doing this process on a production server. Also, the item order in the folder is being lost.

More information

- *How to perform portal_catalog queries*
- <http://webteam.medsci.ox.ac.uk/integrators-developers/transmogrifier-i-want-to-.../>
- <https://github.com/collective/quintagroup.transmogrifier/blob/master/quintagroup/transmogrifier/catalogsource.py>

collective.jsonmigrator

`collective.jsonmigrator` is basically a `collective.transmogrifier` pipeline that pulls Plone content from to JSON views on an old site and writes it into your new site. It's major advantage is that the JSON view product: `collective.jsonify` is very low on dependencies (basically just `simplejson`), it can be installed on old Plone sites that would be difficult if not impossible to install `collective.transmogrifier` into.

See:

- <https://github.com/collective/collective.jsonmigrator>>‘_
- <https://github.com/collective/collective.jsonify>>‘_
- A basic tutorial: <http://www.jowettenterprises.com/blog/plone-content-migration-using-transmogrifier-and-collective.jsonify>>‘_
- <http://stackoverflow.com/questions/13721016/exporting-plone-archetypes-data-in-json>>‘_

Fast content import

For specific use-cases, you can create ‘brains’ first and import later * See [this blog post](#)

Simple JSON export

Below is a simple helper script / BrowserView for a JSON export of Plone folder content. Works Plone 3.3+. It handles also binary data and nested folders.

export.py:

```
"""
    Export folder contents as JSON.

    Can be run as a browser view or command line script.
"""

import os
import base64

try:
    import json
except ImportError:
    # Python 2.54 / Plone 3.3 use simplejson
    # version 2.3.3
    import simplejson as json

from Products.Five.browser import BrowserView
from Products.CMFCore.interfaces import IFolderish
from DateTime import DateTime

#: Private attributes we add to the export list
EXPORT_ATTRIBUTES = ["portal_type", "id"]

#: Do we dump out binary data... default we do, but can be controlled with env var
EXPORT_BINARY = os.getenv("EXPORT_BINARY", None)
if EXPORT_BINARY:
    EXPORT_BINARY = EXPORT_BINARY == "true"
else:
    EXPORT_BINARY = True

class ExportFolderAsJSON(BrowserView):
    """
    Exports the current context folder Archetypes as JSON.
    """
```

```
Returns downloadable JSON from the data.
"""

def convert(self, value):
    """
    Convert value to more JSON friendly format.
    """
    if isinstance(value, DateTime):
        # Zope DateTime
        # https://pypi.python.org/pypi/DateTime/3.0.2
        return value.ISO8601()
    elif hasattr(value, "isBinary") and value.isBinary():

        if not EXPORT_BINARY:
            return None

        # Archetypes FileField and ImageField payloads
        # are binary as OFS.Image.File object
        data = getattr(value.data, "data", None)
        if not data:
            return None
        return base64.b64encode(data)
    else:
        # Passthrough
        return value

def grabArchetypesData(self, obj):
    """
    Export Archetypes schemad data as dictionary object.

    Binary fields are encoded as BASE64.
    """
    data = {}
    for field in obj.Schema().fields():
        name = field.getName()
        value = field.getRaw(obj)
        print "%s" % (value.__class__)

        data[name] = self.convert(value)
    return data

def grabAttributes(self, obj):
    data = {}
    for key in EXPORT_ATTRIBUTES:
        data[key] = self.convert(getattr(obj, key, None))
    return data

def export(self, folder, recursive=False):
    """
    Export content items.

    Possible to do recursively nesting into the children.

    :return: list of dictionaries
    """

    array = []
    for obj in folder.listFolderContents():
```



```

        data = self.grabArchetypesData(obj)
        data.update(self.grabAttributes(obj))

        if recursive:
            if IFolderish.providedBy(obj):
                data["children"] = self.export(obj, True)

        array.append(data)

    return array

def __call__(self):
    """
    """
    folder = self.context.aq_inner
    data = self.export(folder)
    pretty = json.dumps(data, sort_keys=True, indent='    ')
    self.request.response.setHeader("Content-type", "application/json")
    return pretty

def spoof_request(app):
    """
    http://docs.plone.org/develop/plone/misc/commandline.html
    """
    from AccessControl.SecurityManagement import newSecurityManager
    from AccessControl.SecurityManager import setSecurityPolicy
    from Products.CMFCore.tests.base.security import PermissiveSecurityPolicy, _
    ↪OmnipotentUser
    _policy = PermissiveSecurityPolicy()
    setSecurityPolicy(_policy)
    newSecurityManager(None, OmnipotentUser().__of__(app.acl_users))
    return app

def run_export_as_script(path):
    """ Command line helper function.

    Using from the command line::

        bin/instance script export.py yoursiteid/path/to/folder

    If you have a lot of binary data (images) you probably want

        bin/instance script export.py yoursiteid/path/to/folder > yourdata.json

    ... to prevent your terminal being flooded with base64.

    Or just pure data, no binary::

        EXPORT_BINARY=false bin/instance run export.py yoursiteid/path/to/folder

    :param path: Full ZODB path to the folder
    """
    global app

    secure_aware_app = spoof_request(app)
    folder = secure_aware_app.unrestrictedTraverse(path)

```

```
view = ExportFolderAsJSON(folder, None)
data = view.export(folder, recursive=True)
# Pretty pony is prettttyyyyyy
pretty = json.dumps(data, sort_keys=True, indent='    ')
print pretty

# Detect if run as a bin/instance run script
if "app" in globals():
    run_export_as_script(sys.argv[1])
```

Eventish content types

Description

Creating and programming event and eventish content types in Plone

Introduction

Plone supports events as content. Events have a start time, end time and other fields. They can be exported to standard vCal (compatible with Outlook) and iCal (compatible with OSX) formats. A default calendar shows published events in a calendar view.

Note: Recurring events (events repeating with an interval) are supported out-of-the-box on Plone 5.

This is provided by [plone.app.event](#) ([documentation](#))

Purging old events

After the event end day the event stays visible in Plone listings.

You need to have a special janitor script / job if you want to delete old events from your site.

Content rules

- [User manual](#)
- [Developer manual](#)

Archetypes

Archetypes is a subsystem to create content types in Plone 2.x, 3.x and 4.x. Dexterity is replacing it, and is available in 4.1+, becoming the default content type system in Plone 5. Archetypes will remain available through the Plone 5 series.

Fields and widgets

Description

How to read, add, remove and create fields and widgets available for Archetypes content types.

Introduction

This document contains instructions how to manipulate Archetypes schema (data model for content items) and fields and widgets it consists of.

Schema is list of fields associated with a content type. Each field can belong to one *schemata* which corresponds to one Edit tab sub-tab in Plone user interface.

Field schemata is chosen by setting field's `schemata` attribute.

Getting hold of schema objects

Archetypes based data model is defined as Schema object, which is a list of fields.

During application start-up

When your class is being constructed you can refer the schema simply in Python:

```
# Assume you have YourContentSchema object
print YourContentSchema.fields()

class SitsCountry(ATBTreeFolder):
    schema = YourContentSchema

print SitsCountry.schema.fields()
```

During HTTP request processing

You can access context schema object by using `Schema()` accessor.

Note: Run-time schema patching is possible, so `Schema()` output might differ what you put in to your content type during the construction.

Example:

```
schema = context.Schema()
print schema.fields()
```

Schema introspection

How to know what fields are available on content items.

Out of box schema source code

The default Plone schemas are defined

Id and title fields:

- <https://github.com/plone/Products.Archetypes/blob/master/Products/Archetypes/BaseObject.py>

Category and owners schemata: Dublin core metadata

- <https://github.com/plone/Products.Archetypes/blob/master/Products/Archetypes/ExtensibleMetadata.py>

Settings schemata: Exclude from navigation, related items and next/previous navigation

- <https://github.com/plone/Products.ATContentTypes/blob/master/Products/ATContentTypes/content/schemata.py>

Document content

- <https://github.com/plone/Products.ATContentTypes/blob/master/Products/ATContentTypes/content/document.py>

Image content

- <https://github.com/plone/Products.ATContentTypes/blob/master/Products/ATContentTypes/content/image.py>

News content

- <https://github.com/plone/Products.ATContentTypes/blob/master/Products/ATContentTypes/content/newsitem.py>

Run-time introspection

You can get hold of content item schema and its fields as in the example below.

You can do this either in

- *Your own BrowserView Python code*
- *pdb breakpoint*
- *Command line Zope debug console*

Example:

```
for field in context.Schema().fields():
    print "Field:" + str(field) + " value:" + str(field.get(context))
```

Field can be also accessed by name:

```
field = context.Schema()["yourfieldname"]
```

See

- https://github.com/plone/Products.Archetypes/blob/master/Products/Archetypes/Schema/__init__.py

Field name

Field exposes its name through getName() attribute:

```
field = context.Schema()["yourfieldname"]
assert field.getName() == "yourfieldname"
```

Accessing Archetypes field value

Accessor method

Each field has accessor method. Accessor method is

- In your content type class
- Automatically generated if you don't give it manually
- Has name `get + schema field name` with first letter uppercase. E.g. `yourfield` has accessor method `context.getYourfield()` There are a few exceptions to this rule, for fields that correspond to Dublin Core metadata. To conform to the Dublin Core specification, the accessor method for the `title` field is `Title()` and `Description()` for the `description` field.

Raw access

Archetypes has two kinds of access methods:

- normal, `getSomething()`, which filters output;
- raw, the so-called *edit* accessor, `getRawSomething()` which does not filter output.

If you use direct attribute access, i.e. `obj.something` you can get a `BaseUnit` object. `BaseUnit` is an encapsulation of raw data for long text or file. It contains information about mimetype, filename, encoding. To get the raw value of a `BaseUnit` object you can use the `getRaw` method, or more simply `str(baseunit)` (but take care that you don't mess up the encoding).

Indirect access

You can use `field.get(context)` to read values of fields indirectly, without knowing the accessor method.

This example shows how to read and duplicate all values of `lc` object to `nc`:

```
from Products.Archetypes import public as atapi

nc = createObjectSomehow()

# List of field names which we cannot copy
do_not_copy = ["id"]

# Duplicate field data from one object to another
for field in lc.Schema().fields():
    name = field.getName()

    # ComputedFields are handled specially,
    # and UID also
    if not isinstance(field, atapi.ComputedField) and name not in do_not_copy:
        value = field.getRaw(lc)
        newfield = nc.Schema()[name]
        newfield.set(nc, value)
```

```
# Mark creation flag to be set
nc.processForm()
```

Validating objects

Example for *nc* AT object:

```
errors = {}
nc.Schema().validate(nc, None, errors, True, True)
if errors:
    assert not errors, "Got errors:" + str(errors)
```

Checking permissions

`field.writable()` provides a short-cut whether the currently logged in user can change the field value.

Example:

```
field = context.Schema()["phone_number"]
assert field.writable(), "Cannot set phone number"
```

There is also a verbose debugging version which will print the reason to log if the writable condition is not effective:

```
field = context.Schema()["phone_number"]
assert field.writable(debug=True), "Cannot set phone number"
```

Modifying all fields in schema

You might want to modify all schema fields based on some criteria.

Example how to hide all metadata fields:

```
for f in ExperienceEducatorSchema.filterFields(isMetadata=True): f.widget.visible = {
    ↪ "edit" : "invisible" }
```

Reordering fields

See `moveField()` in `Schema/__init__.py`.

Example

```
ProductCardFolderSchema = MountPointSchema.copy() + atapi.Schema((

    # -*- Your Archetypes field definitions here ... -*-
    atapi.StringField(
        'pageTitle',
        stxxxge=atapi.AnnotationStxxxge(),
        widget=atapi.StringWidget(
            label=_("Page title"),
            description=_("Title shown on the page text if differs from the_
    ↪ navigation title"),
```

```

        ),
        default=""
    ),
    ...

))

schemata.finalizeATCTSchema(
    ProductCardFolderSchema,
    folderish=True,
    moveDiscussion=False
)

# Reorder schema fields to the final order,
# show special pageTitle field after actual Title field
ProductCardFolderSchema.moveField("pageTitle", after="title")

```

Hiding widgets

- You should not remove core Plone fields (Title, Description) as they are used by Plone internally e.g. in the navigation tree
- But you can override their accessor functions `Title()` and `Description()`
- You can also hide the widgets

The recommended approach is to hide the widgets, then update the field contents when the relevant data is update. E.g. you can generate title value from fields `firstname` and `lastname`.

Below is an example which uses custom JSON field as input, and then sets title and description based on it:

```

"""Definition of the XXX Researcher content type
"""

import logging
import json # py2.6

from zope.interface import implements, directlyProvides, alsoProvides

from five import grok

from Products.Archetypes.interfaces import IObjectEditedEvent
from Products.Archetypes import atapi
from Products.ATContentTypes.content import folder
from Products.ATContentTypes.content import schemata

from xxx.objects import objectsMessageFactory as _
from xxx.objects.interfaces import IXXXResearcher
from xxx.objects.config import PROJECTNAME

XXXResearcherSchema = folder.ATFolderSchema.copy() + atapi.Schema((

    # -*- Your Archetypes field definitions here ... -*-

```

```
# Stores XXX entry as JSON string
atapi.TextField("XXXData",
                required = True,
                widget=atapi.StringWidget(
                    label="XXX source entry",
                    description="Start typing person's name"
                )),
))

XXXResearcherSchema["title"].widget.visible = {"edit": "invisible" }
XXXResearcherSchema["description"].widget.visible = {"edit": "invisible" }

# Set stxxxge on fields copied from ATFolderSchema, making sure
# they work well with the python bridge properties.

schemata.finalizeATCTSchema(
    XXXResearcherSchema,
    folderish=True,
    moveDiscussion=False
)

class XXXResearcher(folder.ATFolder):
    """A Researcher synchronized from XXX.

    This content will have all

    """
    implements(IXXXResearcher)

    meta_type = "XXXResearcher"
    schema = XXXResearcherSchema

    # -*- Your ATSchema to Python Property Bridges Here ... -*-

    def refreshXXXData(self):
        """
        Performs collective.mountpoint synchronization for one object.
        """
        #synchronize_item(self, logging.WARNING)

    def updateXXX(self, json):
        """
        @param json: JSON payload as a string
        """
        data = self.parseXXXData(json)

        # Set this core Plone fields to actual values,
        # so that we surely co-operate with old legacy code

        title = self.getTitleFromData(data)
        desc = self.getDescriptionFromData(data)

        self.setTitle(title)
        self.setDescription(desc)
```



```

def parseXXXData(self, jsonData):
    """
    @return Python dict
    """
    return json.loads(jsonData)

def getParsedXXXData(self):
    """
    Return XXX JSON data parsed to Python object.
    """

    data = self.getXXXData()
    if data == "" or data is None:
        return None

    return self.parseXXXData(data)

def getTitleFromData(self, data):
    """
    Use lastname + surname from FOAF data as the connt title.
    """

    title = data.get(u"foaf_name", None)

    if title == "" or title is None:
        # Title must have something so that the users
        # can click this item in list...
        title = "(unnamed)"

    # foaf_name is actually list of values, so we need to merge them
    title = " ".join(title)

    return title

def getDescriptionFromData(self, data):
    """ Extract content item description from data blob """

    desc = data.get(u"dc_description", None)

    if desc is None or len(desc) == 0:
        # Decription is not required, we get omit it
        return None

    # dc_description is actually a list of description
    # let's merge them to string here
    desc = " ".join(desc)

    return desc

atapi.registerType(XXXResearcher, PROJECTNAME)

@grok.subscribe(XXXResearcher, IObjectEditedEvent)
def object_edited(context, event):
    """
    Event handler which will update title + description
    values every time the object has been edited.

```

```
@param context: Object for which the event was fired
"""

# Read JSON data entry which user entered on the form
json = context.getXXXData()

if json != None:

    # Update the core fields to reflect changes
    # in JSON data
    context.updateXXX(json)

    # Reflect object changes back to the portal catalog
    # Note that we are running reindexObject()
    # here again... edit itself runs it and
    # we could do some optimization here
    context.reindexObject()
```

Rendering widget

Archetypes is hardwired to render widgets from viewless TAL page templates.

Example how to render widget for field ‘maintext’:

```
<tal:fields tal:define="field_macro here/widgets/field/macros/view;
                        field python:here.Schema()['maintext']">

    <tal:if_visible define="mode string:view;
                        visState python:field.widget.isVisible(here, mode);
                        visCondition python:field.widget.testCondition(context.aq_
→inner.aq_parent, portal, context);"
                        condition="python:visState == 'visible' and visCondition">
        <metal:use_field use-macro="field_macro" />
    </tal:if_visible>
</tal:fields>
```

Creating your own Field

Here is an example how to create a custom field based on TextField.

Example (mfabrik/rstpage/archetypes/fields.py):

```
from Products.Archetypes import public as atapi
from Products.Archetypes.Field import TextField, ObjectField, encode, decode,
→registerField

from mfabrik.rstpage.transform import transform_rst_to_html

class RSTField(atapi.TextField):
    """ """

    def _getCooked(self, instance, text):
        """ Perform reST to HTML transformation for the field content.

        """
```

```

        html, errors = transform_rst_to_html(text)
        return html

    def get(self, instance, **kwargs):
        """ Field accessor.

        Define view mode accessor for the widget.

        @param instance: Archetypes content item instance

        @param kwargs: Arbitrary parameters passed to the field getter
        """

        # Read the stored field value from the instance
        text = ObjectField.get(self, instance, **kwargs)

        # raw = edit mode, get reST source in that case
        raw = kwargs.get("raw", False)

        if raw:
            # Return reST source
            return text
        else:
            # Return HTML for viewing
            return self._getCooked(instance, text)

registerField(RSTField,
             title='Restructured Text field',
             description=('Edit HTML as reST source'))

```

Automatically generating description based on body text

Below is a through-the-web (TTW) Python Script which you can drop into through the Management Interface.

Use case: People are lazy to write descriptions (as in Dublin Core metadata). You can generate some kind of description by taking the few first sentences of the text.

This is not perfect, but this is way better than empty description.

This script will provide one-time operation to automatically generate content item descriptions based on their body text by taking the first three sentences.

The script will provide logging output to standard Plone log (var/log and stdout if Plone is run in debug mode).

Example code:

```

def create_automatic_description(content, text_field_name="text"):
    """ Creates an automatic description from HTML body by taking three first
    ↪ sentences.

    Takes the body text

    @param content: Any Plone contentish item (they all have description)

    @param text_field_name: Which schema field is used to supply the body text (may
    ↪ very depending on the content type)
    """

```

```
# Body is Archetype "text" field in schema by default.
# Accessor can take the desired format as a mimetype parameter.
# The line below should trigger conversion from text/html -> text/plain,
↳ automatically using portal_transforms
field = content.Schema()[text_field_name]

# Returns a Python method which you can call to get field's
# for a certain content type. This is also security aware
# and does not breach field-level security provided by Archetypes
accessor = field.getAccessor(content)

# body is UTF-8
body = accessor(mimetype="text/plain")

# Now let's take three first sentences or the whole content of body
sentences = body.split(".")

if len(sentences) > 3:
    intro = ".".join(sentences[0:3])
    intro += "." # Don't forget closing the last sentence
else:
    # Body text is shorter than 3 sentences
    intro = body

content.setDescription(intro)

# context is the reference of the folder where this script is run
for id, item in context.contentItems():
    # Iterate through all content items (this ignores Zope objects like this script,
    ↳ itself)

    # Use RestrictedPython safe logging.
    # plone_log() method is permission aware and available on any contentish object
    # so we can safely use it from through-the-web scripts
    context.plone_log("Fixing:" + id)

    # Check that the description has never been saved (None)
    # or it is empty, so we do not override a description someone has
    # set before automatically or manually
    desc = context.Description() # All Archetypes accessor method, returns UTF-8,
    ↳ encoded string

    if desc is None or desc.strip() == "":
        # We use the HTML of field called "text" to generate the description
        create_automatic_description(item, "text")

# This will be printed in the browser when the script completes successfully
return "OK"
```

See also

- <http://blog.mfabrik.com/2010/06/04/automatically-generating-description-based-on-body-text/>

Vocabularies

Archetypes has its own vocabulary infrastructure which is not compatible with *zope.schema vocabularies*.

Dynamic vocabularies

- <http://www.universalwebservices.net/web-programming-resources/zope-plone/dynamic-vocabularies-in-plone-archetypes>

Rendering single field

Example:

```
<metal:fieldMacro use-macro="python:context.widget(field.getName(), mode='edit')" />
```

Hiding widgets conditionally

AT widgets have condition *expression*.

Example how to set a condition for multiple widgets to call a BrowserView to ask whether the widget should be visible or not:

```
for field in ResearcherSchema.values():
    # setCondition() is in Products.Archetypes.Widget
    # possible expression variables are_ object, portal, folder.
    field.widget.setCondition("python:object.restrictedTraverse('@@msd_widget_
    ↪condition')('" + field.getName() + "')")
```

The related view with some sample code:

```
class WidgetCondition(BrowserView):
    """
    This is referred in msd.researcher schema conditions field.
    """

    def __call__(self, fieldName):
        """
        """
        settings = getResearcherSettings(self.context)
        customization = settings.getFieldCustomization(fieldName, "visible")
        if customization is not None:
            return customization

        # Default is visible
        return True
```

Dynamic field definitions

You can override `Schema()` and `Schemata()` methods in your content type class to poke the schema per HTTP request access basis.

Example:

```
def Schema(self):
    """ Overrides field definitions in fly.

    """

    # XXX: Cache this method?
    from Acquisition import ImplicitAcquisitionWrapper
    from Products.Archetypes.interfaces import ISchema

    # Create modifiable copy of schema
    # See Products.Archetypes.BaseObject
    schema = ISchema(self)
    schema = schema.copy()
    schema = ImplicitAcquisitionWrapper(schema, self)

    settings = self.getResearchSettings()

    for row in settings.getFieldCustomizations():
        name = row.get("fieldName", None)
        vocab = row.get("vocabToUse", None)

        field = schema.get(name, None)

        if field and vocab and hasattr(field, "vocabulary"):
            # Modify field copy ion

            displayList = settings.getVocabulary(vocab)
            if displayList is not None:
                field.vocabulary = displayList

    return schema
```

Field storages

Field storage tells how the value of schema field is stored.

AttributeStorage

Products.Archetypes.storage.AttributeStorage

This is recommended for data which is *always* read when the object is accessed: title, description, etc.

AnnotationStorage

Products.Archetypes.storage.annotation.AnnotationStorage

AnnotationStorage creates an object attribute `__annotations__` which is an OOBTree object. An OOBTree uses *buckets* as the smallest persistent entity. A bucket usually holds a small number of items. Buckets are loaded on request and as needed compared to using native Python datatypes.

It is safe to assume that you can fit few variables to one bucket.

You also might want to define `ATFieldProperty` accessor if you are using this storage. This allows you to read the object value using standard Python attribute access notation.

Note that in this case the access goes through AT accessor and mutator functions. This differs from raw storage value access: for example the AT accessor encodes strings to UTF-8 before returning them.

Example:

```
VariantProductSchema['myField'].storage = atapi.AnnotationStorage()

class VariantProduct(folder.ATFolder):

    meta_type = "VariantProduct"
    schema = VariantProductSchema

    myField = atapi.ATFieldProperty('title')

product = VariantProduct()

product.setMyField("foobar") # Set field using AT mutator method

products.myField = # AT field property magic. This is equal to product.getMyField()
```

SQLStorage

This stores field values in an external SQL database.

- [An old documentation how to use SQL storage.](#)

FSSStorage

Store the raw values of fields on the file system.

```
# Usual Zope/CMF/Plone/Archetypes imports
...
from iw.fss.FileSystemStorage import FileSystemStorage
...
my_schema = Schema((
    FileField('file',
        ...
        storage=FileSystemStorage(),
        widget=FileWidget(...),
    ),
    ...
))
...
```

- [Official documentation of fss.](#)

Archetypes ReferenceFields

Description

Using ReferenceField to have references to other Archetypes content items in Plone.

Introduction

Archetypes comes with a kind of field called ReferenceField which is used to store references to other Archetypes objects, or any object providing the IReferenceable interface.

References are maintained in the `uid_catalog` and `reference_catalog` catalogs. You can find both at the root of your Plone site. Check them to see their indexes and metadata.

Although you could use the ZCatalog API to manage Archetypes references, these catalogs are rarely used directly. A ReferenceField and its API is used instead.

Example declaration of a ReferenceField inside a schema:

```
MyCTSchema = atapi.Schema((
    ...
    atapi.ReferenceField('myReferenceField',
        relationship = 'somerelationship',
    ),
    ...
))
```

Check the *Fields Reference* section in the *Archetypes Developer Manual* at <https://plone.org> to learn about the ReferenceField available options.

Archetypes reference fields just store the UID (Universal Object Identifier) of an object providing the IReferenceable interface. Continuing with the example above, you will usually use the regular field API (getters/setters).

Get the UID of a referenceable object:

```
>>> referenceableobject_uid = referenceableobject.UID()
```

To set a reference, you can use the the setter method with either a list of UIDs or one UID string, or one object or a list of objects (in the case the ReferenceField is multi-valued) to which you want to add a reference to. `None` and `[]` are equal.

In this example we set a reference from the `myct1` object to the `referenceableobject` object:

```
>>> myct1.setMyReferenceField(referenceableobject_uid)
```

To get the object(s) referenced, just use the getter. Note that what you get is the objects themselves, not their “*brains*”:

```
.. TODO:: Add a glossary entry for brains.
```

More info in Varnish section of this manual.

```
>>> myct1.getMyReferenceField() == referenceableobject
True
```

Customizing editing interface

Remove metadata tabs

Remove *Manage properties* permissions from the users who should not see metadata fields. Do this for all fields under the schema.

DataGridField

This document contains miscellaneous notes about `DataGridField` field and `DataGridWidget` widget.

`DataGridField` is an Archetypes field and widget to add tabular structures to your custom content types.

Basics

`DataGridField` acts as any other Archetypes based field.

To read DGF content use *accessor* function:

```
data = myobject.getMyDGF() #
```

Data is a Python list of dictionaries. Each dictionary presents one row. Dictionary keys are column ids and dictionary values are cell values.

To set DGF content you must replace all rows at once:

```
mydata = [
    { "column1" : "value", "column2" : "something else" }, # row 1
    { "column1" : "xxx", "column2" : "yyy" } # row 2
]

context.setMyDGF(mydata)
```

To append a row to DFG, you need to read it, manipulate the list, and then reset the value:

```
rows = myobject.getMyDGF() # This returns a copy which you can modify freely
rows.append({ "column1" : "value", "column2" : "something else" })
myobject.setMyDGF(rows) # Now set the value with one new row
```

Modify cell value in DGF:

```
rows = myobject.getMyDGF() # This returns a copy which you can modify freely
rows[0]["column1"] = "newvalue" # Set a string value for row 1, cell 1 (cell using ↵
↵column id column1)
myobject.setMyDGF(rows) # Now set the value with one new row
```

CheckboxColumn

Checkbox column values are handled specially:

```
def convertCheckboxValue(value):
    """ DataGridField value converter for CheckboxColumn """
    if value is None:
        return None

    if value == '':
```

```
        return False

    if value == '1':
        return True

    # XXX: Not sure if happens
    if value == '0':
        return False

    raise RuntimeError("Bad checkbox value:" + value)
```

Other resources

Please enable DEBUG in <https://github.com/collective/Products.DataGridField/blob/master/Products/DataGridField/config.py> on your local computer. After this setting has been changed, you can run unit tests and install example types on your computer.

Refer [unit tests](#) for more code examples.

Refer [Archetypes manual](#) for basics Archetypes developer information.

Validators

Introduction

This page has tips how to validate fields defined in Archetypes schema.

List of default validators

- <https://github.com/plone/Products.validation/blob/master/Products/validation/validators/BaseValidators.py>

Creating a validator

A custom validator should return True if valid, or an error string if validation fails. This is especially important to remember when chaining validators together. See the tutorials below for further details:

- <http://play.pixelblaster.ro/blog/archive/2006/08/27/creating-an-archetypes-validator>
- <http://www.pererikstrandberg.se/blog/index.cgi?page=PloneArchetypesFieldValidator>

Files

Description

Using files with Archetype field

local

- *Files*
 - *Download URL for files for ATFile content*
 - *Checking whether a File field has uploaded content*
 - *Setting max file size to FileField and ImageField*

Download URL for files for ATFile content

Append @@download view to URL.

Checking whether a File field has uploaded content

Calling AT File field accessor will return a File object:

```
(Pdb) self.context.getAttachment()
<File at /mfabrik/success-stories/case-studies/finnish-national-broadcasting-company/
↪attachment>
```

Note that this may return None if the content item has been constructed but the form has not been properly saved.

If the size is 0, the file is not yet uploaded:

```
(Pdb) attach.getSize()
0
```

Example how to check in a view whether AT context file size exists:

```
@property
def available(self):

    # Make sure that we have content item of right kind
    if ICaseStudy.providedBy(self.context):

        # Make sure the content item is not anymore in the creation stage
        if self.context.getAttachment() is not None:

            # Check the content of File field
            if self.context.getAttachment().getSize() > 0:
                return True

        return False
```

Setting max file size to FileField and ImageField

TODO

<http://stackoverflow.com/questions/11347200/setting-max-upload-size-for-archetypes-filefield>

Old, deprecated, info

- <http://keeshink.blogspot.fi/2009/09/how-to-limit-file-upload-size.html>

Converting one Content Type into another

Description

It is possible to ‘convert’ one content type into another by extracting content from the source content type and adding it to the new content type.

local

- *Converting one Content Type into another*
 - *Converting Pages into News Items*
 - *Converting Images into News Items*

Converting Pages into News Items

In this example we take a folder of *Pages* (meta type: Document) and create *News Items* from them:

```
"""
from the News Item type, the new items will be content copies
of their corresponding Pages (Documents) """

source_contenttype = 'Document'
target_contenttype = 'Service'

items = context.listFolderContents(
    contentFilter={"portal_type": source_contenttype})

for item in items:
    id = "%s-new" % item.getId()
    title = item.Title()
    description = item.Description()
    text = item.getText()

    service = context.invokeFactory(target_contenttype, id,
                                    title=title,description=description,text=text)
```

Converting Images into News Items

This is similar to the example of converting pages into news items. Notice that when we pass the image data to `invokeFactory` we need to make it into a string:

```
source_contenttype = 'Image'
target_contenttype = 'News Item'

items = context.listFolderContents(
    contentFilter={"portal_type": source_contenttype})

for item in items:
    id = "%s-new" % item.getId()
    title = item.Title()
```

```
imageCaption = text = description = item.Description()
image = str(item.getImage())

service = context.invokeFactory(
    target_contenttype,
    id,
    title=title,
    description=description,
    imageCaption=imageCaption,
    text=text,
    image=image)
```

Templates

Description

Overriding templates with Archetypes

local

- *Templates*
 - *Introduction*
 - *The template loading mechanism*

Introduction

This document will tell how to create custom templates for Plone and Archetypes based content.

This does not deal with

- *browser views*
- *generic old style template overrides*

The template loading mechanism

Archetypes tries to look up a template with name

- *Content type name lowercased + _view.pt*
- *Content type name lowercased + _edit.cpt*

from portal_skins.

Example controlled page template (cpt) file yourcontenttype.cpt:

Check More info links

For cpt files (controlled page template) you'll also need corresponding .metadata file:

```
[default]
title = Edit Your Content Type

[validators]
validators = validate_atct
validators..form_add =
validators..cancel =

[actions]
action.success = traverse_to:string:content_edit
action.success..cancel = redirect_to:python:object.REQUEST['last_referer']
action.success_add_reference = redirect_to:python:object.REQUEST['last_referer']
action.failure = traverse_to_action:string:edit
action.next_schemata = traverse_to_action:string:edit
```

More info

- <https://plone.org/documentation/manual/theme-reference/buildingblocks/skin/templates/how-to-customise-view-or-edit-on-archetypes-content-items>

References

Description

Inter-content references in Plone are done using the `reference_catalog` tool.

Introduction

Plone uses a persistent tool called `reference_catalog` to store (Archetypes) object references. It is used by the out-of-the-box “Related items” and you can use it in your own content types with `ReferenceField`.

`reference_catalog` references can be bidirectional.

The `reference_catalog` is a catalog just like the *portal_catalog* — it just uses different indexes and metadata.

The `reference_catalog` is defined in `ReferenceEngine.py`.

Using references

Here is an example how to use reference field to make *programme* -> *researcher* references, and how to do reverse look-ups for the relationship.

You use `getReferences()` and `getBackReferences()` methods to look up relationships.

Example:

```
from Products.CMFCore.utils import getToolByName
from Products.Archetypes.config import REFERENCE_CATALOG

def getResearcherProgrammes(researcher):
    """
    Find all Programmes which refer to this researcher.

    The Programme<->Researcher relationship is defined in Programme as::
```

```

    atapi.ReferenceField(
        name='researchers',
        widget=ReferenceBrowserWidget(
            label="Researchers",
            description="Researchers involved in this project",
            base_query={'object_provides': IResearcher.__identifier__ },
            allow_browse=0,
            show_results_without_query=1,
        ),
        multiValued=1,
        relationship="researchers_in_theme"
    ),

    @param researcher: Content item on the site
    """
    reference_catalog = getToolByName(researcher, REFERENCE_CATALOG)

    # relationship: field name used
    # Plone 4.1: objects=True argument to fetch full objects, not just
    # index brains
    references = reference_catalog.getBackReferences(
        researcher,
        relationship="researchers_in_theme")
    # Resolve Reference objects to full objects
    # Return a generator method which will yield all full objects
    return [ ref.getSourceObject() for ref in references ]

```

Dexterity

Description

Dexterity content subsystem for Plone: info for the developers.

Introduction

Dexterity is a subsystem for content objects. It is the standard content type for Plone 5 and onward and can be already used with Plone 4.

- Dexterity developer manual
- How Dexterity is related to Archetypes

mr.bob templates

Please see [bobtemplates.plone](#) page for project skeleton templates for Dexterity.

Content creation permissions

Please read dexterity and permissions

Exclusion from navigation

This must be enabled separately for Dexterity content types with a behavior.

```
<property name="behaviors">
  <element value="plone.app.content.interfaces.INameFromTitle" />
  <element value="plone.app.dexterity.behaviors.metadata.IBasic"/>
  <element value="plone.app.dexterity.behaviors.exclfromnav.IExcludeFromNavigation"/
  </>
</property>
```

Then you can manually also check this property:

```
for t in self.tabs:
    nav = None
    try:
        nav = IExcludeFromNavigation(t)
    except:
        pass
    if nav:
        if nav.exclude_from_nav == True:
            # FAQ page - do not show in tabs
            continue
```

Custom add form/view

Dexterity relies on `++add++yourcontent.type.name` traverser hook defined in `Products/CMFCore/namespace.py`.

It will look up a multi-adapter using this expression:

```
if ti is not None:
    add_view = queryMultiAdapter((self.context, self.request, ti),
                                name=ti.factory)

    if add_view is None:
        add_view = queryMultiAdapter((self.context, self.request, ti))
```

The name parameter is the `portal_types` id of your content type.

You can register such an adapter in `configure.zcml`

```
<configure
  xmlns="http://namespaces.zope.org/zope"
  xmlns:browser="http://namespaces.zope.org/browser"
>

  <adapter
    for="Products.CMFCore.interfaces.IFolderish
         plone.dexterity.interfaces.IDexterityFTI"
    provides="zope.publisher.interfaces.browser.IBrowserPage"
    factory=".flexicontent.AddView"
    name="your.app.flexiblecontent"
  />

</configure>
```

Then you can inherit from the proper `plone.dexterity` base classes:


```

from plone.dexterity.browser.add import DefaultAddForm, DefaultAddView

class AddForm(DefaultAddForm):

    def update(self):
        DefaultAddForm.update(self)

    def updateWidgets(self):
        """ """
        # Some custom code here

    def getBlockPlanJSON():
        return getBlockPlanJSON()

class AddView(DefaultAddView):
    form = AddForm

```

See also:

- *FTI*
- *z3c.form*

Custom edit form

Example:

```

from five import grok
from plone.directives import dexterity

class EditForm(dexterity.EditForm):

    grok.context(IFlexibleContent)

    def updateWidgets(self):
        """ """
        dexterity.EditForm.updateWidgets(self)

        # XXX: customize widgets here

```

Registering an edit form works by registering a normal browser page.

```

<configure
  xmlns="http://namespaces.zope.org/zope"
  xmlns:browser="http://namespaces.zope.org/browser"
  >

  <browser:page
    for="your.app.flexiblecontent"
    class=".flexicontent.EditView"
    name="edit"
    />

</configure>

```

In the example above it is important, that you give the browser page the name “edit”.

Models, forms, fields and widgets

Plone includes several alternative form mechanisms:

For content-oriented forms:

- *Dexterity* for Plone 4.1+
- *Archetypes* was used for content types in Plone 3.x, but can still be used in Plone 4 and 5 for migrated sites. For any new development, Dexterity is **strongly** recommended.

For convenience forms built and maintained through-the-web and where the results are stored in CSV sheet or emailed:

- *PloneFormGen*

For application and utility forms where custom logic is added by writing Python code:

- `z3c.form` for Plone 4.x
- `zope.formlib` was used for stock forms in Plone 3.x

This documentation applies only for form libraries.

You need to identify which form library you are dealing with and read the form library specific documentation.

Zope 3 schema (`zope.schema` package) is database-neutral and framework-neutral way to describe Python data models.

Modelling data

Modelling using `zope.schema`

Description

`zope.schema` package provide a storage-neutral way to define Python object models with validators.

Introduction

Zope 3 schemas are a database-neutral and form-library-neutral way to describe Python data models.

Plone uses Zope 3 schemas for these purposes:

- to describe persistent data models;
- to describe HTML form data;
- to describe ZCML configuration data.

Since Zope 3 schemas are not bound to e.g. a SQL database engine, it gives you very reusable way to define data models.

Schemas are just regular Python classes, with some special attribute declarations. They are always subclasses of `zope.interface.Interface`. The schema itself cannot be a concrete object instance — you need to either have a `persistent.Persistent` object (for database data) or a `z3c.form.form.Form` object (for HTML forms).

Zope 3 schemas are used for tasks like:

- defining allowed input data format (string, integer, object, list, etc.) for Python class instance attributes;
- specifying required attributes on an object;

- defining custom validators on input data.

The basic unit of data model declaration is the *field*, which specifies what kind of data each Python attribute can hold.

More info

- [zope.schema](#) on PyPi
- [zope.schema source code](#) - definite source for field types and usage.

`zope.schema` provides a very comprehensive set of fields out of the box. Finding good documentation for them, however, can be challenging. Here are some starting points:

- [Dexterity field list](#).

Example of a schema

Let's define a data model to store addresses:

```
import zope.interface
from zope import schema

class ICheckoutAddress(zope.interface.Interface):
    """ Provide meaningful address information.

    This is not 1:1 with getpaid.core interfaces, but
    more like a better guess.
    """

    first_name = schema.TextLine(title=_("First name"), default=u"")
    last_name = schema.TextLine(title=_("Last name"), default=u"")
    organization = schema.TextLine(title=_("Organization"), default=u"")
    phone = schema.TextLine(title=_("Phone number"), default=u"")
    country = schema.Choice(title=_("Country"),
        vocabulary="getpaid.countries", required=False, default=None)
    state = schema.Choice(title=_("State"),
        vocabulary="getpaid.states", required=False, default=None)
    city = schema.TextLine(title=_("City"), default=u"")
    postal_code = schema.TextLine(title=_("Postal code"), default=u"")
    street_address = schema.TextLine(title=_("Address"), default=u")
```

Next, we define a concrete persistent class which uses this data model. We can use this class to store data based on our model definition in the ZODB database.

We use `zope.schema.fieldproperty.FieldProperty` to bind persistent class attributes to the data definition.

Example:

```
from persistent import Persistent # Automagical ZODB persistent object
from zope.schema.fieldproperty import FieldProperty

class CheckoutAddress(Persistent):
    """ Store checkout address """

    # Declare that all instances of this class will
    # conform to the ICheckoutAddress data model:
    zope.interface.implements(ICheckoutAddress)
```

```
# Provide the fields:
first_name = FieldProperty(ICheckoutAddress["first_name"])
last_name = FieldProperty(ICheckoutAddress["last_name"])
organization = FieldProperty(ICheckoutAddress["organization"])
phone = FieldProperty(ICheckoutAddress["phone"])
country = FieldProperty(ICheckoutAddress["country"])
state = FieldProperty(ICheckoutAddress["state"])
city = FieldProperty(ICheckoutAddress["phone"])
postal_code = FieldProperty(ICheckoutAddress["postal_code"])
street_address = FieldProperty(ICheckoutAddress["street_address"])
```

For persistent objects, see *[persistent object documentation](#)*.

Using schemas as data models

Based on the example data model above, we can use it in e.g. content type *[browser views](#)* to store arbitrary data as content type attributes.

Example:

```
class MyView(BrowserView):
    """ Connect this view to your content type using a ZCML declaration.
    """

    def __call__(self):
        # Get the content item which this view was invoked on:
        context = self.context.aq_inner

        # Store a new address in it as the ``test_address`` attribute
        context.test_address = CheckoutAddress()
        context.test_address.first_name = u"Mikko"
        context.test_address.last_name = u"Ohtamaa"

        # Note that you can still add arbitrary attributes to any
        # persistent object. They are simply not validated, as they
        # don't go through the ``zope.schema`` FieldProperty
        # declarations.
        # Do not do this, you will regret it later.
        context.test_address.arbitrary_attribute = u"Don't do this!"
```

Field constructor parameters

The `Field` base class defines a list of standard parameters that you can use to construct schema fields. Each subclass of `Field` will have its own set of possible parameters in addition to this.

See the full list [here](#).

Title field title as unicode string

Description field description as unicode string

required boolean, whether the field is required

default Default value if the attribute is not present

... and so on.

Warning: Do not initialize any non-primitive values using the *default* keyword parameter of schema fields. Python and the ZODB stores objects by reference. Python code will construct only *one* field value during schema construction, and share its content across all objects. This is probably not what you intend. Instead, initialize objects in the `__init__()` method of your schema implementer.

In particular, dangerous defaults are: `default=[]`, `default={}`, `default=SomeObject()`.

Schema introspection

The `zope.schema._schema` module provides some introspection functions:

- `getFieldNames(schema_class)`
- `getFields(schema_class)`
- `getFieldNamesInOrder(schema)` — retain the original field declaration order.
- `getFieldsInOrder(schema)` — retain the original field declaration order.

Example:

```
import zope.schema
import zope.interface

class IMyInterface(zope.interface.Interface):

    text = zope.schema.TextLine()

# Get list of schema fields from IMyInterface
fields = zope.schema.getFields(IMyInterface)
```

Dumping schema data

Below is an example how to extract all schema defined fields from an object.

```
from collections import OrderedDict

import zope.schema

def dump_schemed_data(obj):
    """
    Prints out object variables as defined by its zope.schema Interface.
    """
    out = OrderedDict()

    # Check all interfaces provided by the object
    ifaces = obj.__provides__.__iro__

    # Check fields from all interfaces
    for iface in ifaces:
        fields = zope.schema.getFieldsInOrder(iface)
        for name, field in fields:
```

```
        # ('header', <zope.schema._bootstrapfields.TextLine object at 0x1149dd690>
        ↪)
        out[name] = getattr(obj, name, None)

    return out
```

Finding the schema for a Dexterity type

When trying to introspect a Dexterity type, you can get a reference to the schema thus:

```
from zope.component import getUtility
from plone.dexterity.interfaces import IDexterityFTI

schema = getUtility(IDexterityFTI, name=PORTAL_TYPE_NAME).lookupSchema()
```

...and then inspect it using the methods above. Note this won't have behavior fields added to it at this stage, only the fields directly defined in your schema.

Field order

The `order` attribute can be used to determine the order in which fields in a schema were defined. If one field was created after another (in the same thread), the value of `order` will be greater.

Default values

To make default values of schema effective, class attributes must be implemented using `FieldProperty`.

Example:

```
import zope.interface
from zope import schema
from zope.schema.fieldproperty import FieldProperty

class ISomething(zope.interface.Interface):
    """ Sample schema """
    some_value = schema.Bool(default=True)

class SomeStorage(object):

    some_value = FieldProperty(ISomething["some_value"])

something = SomeStorage()
assert something.some_value == True
```

Validation and type constrains

Schema objects using field properties provide automatic validation facilities, preventing setting badly formatted attributes.

There are two aspects to validation:

- Checking the type constraints (done automatically).
- Checking whether the value fills certain constraints (validation).

Example of how type constraints work:

```
class ICheckoutData(zope.interface.Interface):
    """ This interface defines all the checkout data we have.

    It will also contain the ``billing_address``.
    """

    email = schema.TextLine(title=_("Email"), default=u"")

class CheckoutData(Persistent):

    zope.interface.implements(ICheckoutData)

    email = FieldProperty(ICheckoutData["email"])

def test_store_bad_email(self):
    """ Check that we can't put data to checkout """

    data = getpaid.expercash.data.CheckoutData()

    from zope.schema.interfaces import WrongContainedType, WrongType, NotUnique

    try:
        data.email = 123 # Can't set email field to an integer.
        raise AssertionError("Should never be reached.")
    except WrongType:
        pass
```

Example of validation (email field):

```
from zope import schema

class InvalidEmailError(schema.ValidationError):
    __doc__ = u'Please enter a valid e-mail address.'

def isEmail(value):
    if re.match('^[^@]+EMAIL_RE', value):
        return True
    raise InvalidEmailError

class IContact(Interface):
    email = schema.TextLine(title=u'Email', constraint=isEmail)
```

Persistent objects and schema

ZODB persistent objects do not provide facilities for setting field defaults or validating the data input.

When you create a persistent class, you need to provide field properties for it, which will sanify the incoming and outgoing data.

When the persistent object is created it has no attributes. When you try to access the attribute through a named `zope.schema.fieldproperty.FieldProperty` accessor, it first checks whether the attribute exists. If the attribute is not there, it is created and the default value is returned.

Example:

```
from persistent import Persistent
from zope import schema
from zope.interface import implements, alsoProvides
from zope.component import adapts
from zope.schema.fieldproperty import FieldProperty

# ... other implementation code ...

class IHeaderBehavior(form.Schema):
    """ Sample schema """
    inheritable = schema.Bool(
        title=u"Inherit header",
        description=u"This header is visible on child content",
        required=False,
        default=False)

    block_parents = schema.Bool(
        title=u"Block parent headers",
        description=u"Do not show parent headers for this content",
        required=False,
        default=False)

    # Contains list of HeaderAnimation objects
    alternatives = schema.List(
        title=u"Available headers and animations",
        description=u"Headers and animations uploaded here",
        required=False,
        value_type=schema.Object(IHeaderAnimation))

alsoProvides(IHeaderAnimation, form.IFormFieldProvider)

class HeaderBehavior(Persistent):
    """ Sample persistent object for the schema """

    implements(IHeaderBehavior)

    #
    # zope.schema magic happens here - see FieldProperty!
    #

    # We need to declare field properties so that objects will
    # have input data validation and default values taken from schema
    # above

    inheritable = FieldProperty(IHeaderBehavior["inheritable"])
    block_parents = FieldProperty(IHeaderBehavior["block_parents"])
    alternatives = FieldProperty(IHeaderBehavior["alternatives"])
```

Now you see the magic:


```
header = HeaderBehavior()
# This triggers the ``alternatives`` accessor, which returns the default
# value, which is an empty list
assert header.alternatives == []
```

Collections (and multichoice fields)

Collections are fields composed of several other fields. Collections also act as multi-choice fields.

For more information see:

- Using Zope schemas with a complex vocabulary and multi-select fields
- Collections section in `zope.schema` documentation
- Schema field sources documentation
- Choice field
- List field.

Single-choice example

Only one value can be chosen.

Below is code to create Python logging level choice:

```
import logging

from zope.schema.vocabulary import SimpleVocabulary, SimpleTerm

def _createLoggingVocabulary():
    """ Create zope.schema vocabulary from Python logging levels.

    Note that term.value is int, not string.

    _levelNames looks like::

        {0: 'NOTSET', 'INFO': 20, 'WARNING': 30, 40: 'ERROR', 10: 'DEBUG', 'WARN': 30,
→ 50:
        'CRITICAL', 'CRITICAL': 50, 20: 'INFO', 'ERROR': 40, 'DEBUG': 10, 'NOTSET': 0,
→ 30: 'WARNING'}
```

@return: Iterable of SimpleTerm objects
 """
 for level, name in logging._levelNames.items():

 # logging._levelNames dictionary is bidirectional, let's
 # get numeric keys only

 if type(level) == int:
 term = SimpleTerm(value=level, token=str(level), title=name)
 yield term

Construct SimpleVocabulary objects of log level -> name mappings
logging_vocabulary = SimpleVocabulary(list(_createLoggingVocabulary()))

```
class ISyncRunOptions (Interface):

    log_level = schema.Choice (vocabulary=logging_vocabulary,
                               title=u"Log level",
                               description=u"One of python logging module constants",
                               default=logging.INFO)
```

Multi-choice example

Using `zope.schema.List`, many values can be chosen once. Each value is atomically constrained by *value_type* schema field.

Example:

```
from zope import schema
from plone.supermodel import model
from plone.autoform import directives as form

from z3c.form.browser.checkbox import CheckBoxFieldWidget

class IMultiChoice (model.Schema):
    ...

    # Contains lists of values from Choice list using special "get_field_list"
    ↪ vocabulary
    # We also give a plone.autoform.directives hint to render this as
    # multiple checkbox choices
    form.widget (yourField=CheckBoxFieldWidget)
    yourField = schema.List (title=u"Available headers and animations",
                             description=u"Headers and animations uploaded here",
                             required=False,
                             value_type=schema.
    ↪ Choice (source=yourVocabularyFunction),
                             )
```

Dynamic schemas

Schemas are singletons, as there only exist one class instance per Python run-time. For example, if you need to feed schemas generated dynamically to form engine, you need to

- If the form engine (e.g. `z3c.form` refers to schema fields, then replace these references with dynamically generated copies)
- Generate a Python class dynamically. Output Python source code, then `eval()` it. Using `eval()` is almost always considered as a bad practice.

Warning: Though it is possible, you should not modify `zope.schema` classes in-place as the same copy is shared between different threads and if there are two concurrent HTTP requests problems occur.

Replacing schema fields with dynamically modified copies

The below is an example for `z3c.form`. It uses Python `copy` module to copy `f.field` reference, which points to `zope.schema` field. For this field copy, we modify *required* attribute based on input.

Example:

```
@property
def fields(self):
    """ Get the field definition for this form.

    Form class's fields attribute does not have to
    be fixed, it can be property also.
    """

    # Construct the Fields instance as we would
    # normally do in more static way
    fields = z3c.form.field.Fields(ICheckoutAddress)

    # We need to override the actual required from the
    # schema field which is a little tricky.
    # Schema fields are shared between instances
    # by default, so we need to create a copy of it
    if self.optional:
        for f in fields.values():
            # Create copy of a schema field
            # and force it unrequired
            schema_field = copy.copy(f.field) # shallow copy of an instance
            schema_field.required = False
            f.field = schema_field
```

Don't use dict {} or list [] as a default value

Because how Python object construction works, giving [] or {} as a default value will make all created field values to share this same object.

<http://effbot.org/zone/default-values.htm>

Use value adapters instead

- <https://pypi.python.org/pypi/plone.directives.form#value-adapters>

Vocabularies

Description

Vocabularies are lists of (value -> human readable title) pairs used by e.g. selection drop downs. `zope.schema` provides tools to programmatically construct their vocabularies.

Introduction

Vocabularies specify options for choice fields.

Vocabularies are normally described using `zope.schema.vocabulary.SimpleVocabulary` and `zope.schema.vocabulary.SimpleTerm` objects. [See the source code](#).

Vocabulary terms

`zope.schema` defines different vocabulary term possibilities.

A term is an entry in the vocabulary. The term has a value. Most terms are tokenised terms which also have a token. Some terms are titled, meaning they have a title that is different to the token.

`SimpleTerm` instances

`SimpleTerm.token` must be an ASCII string. It is the value passed with the request when the form is submitted. A token must uniquely identify a term.

`SimpleTerm.value` is the actual value stored on the object. This is not passed to the browser or used in the form. The value is often a unicode string, but can be any type of object.

`SimpleTerm.title` is a unicode string or translatable message. It is used for display in the form.

For further details please read the [interfaces specification](#)

Note: If you need international texts please note that only title is, and should be, translated. Value and token must always carry the same value.

Creating a vocabulary

Example 1 - form a list of tuples:

```
from zope.schema.vocabulary import SimpleTerm
from zope.schema.vocabulary import SimpleVocabulary

items = [ ('value1', u'This is label for item'), ('value2', u'This is label for value_
↪2')]
terms = [ SimpleTerm(value=pair[0], token=pair[0], title=pair[1]) for pair in items ]

myVocabulary = SimpleVocabulary(terms)
```

Example 2 - using the vocabulary from (1):

```
from plone.supermodel import form
from zope import schema

class ISampleSchema(form.Schema):

    contentMedias = schema.Choice(
        title=u'Test choice'
        vocabulary=myVocabulary,
    )
```

Example 3 - create a vocabulary from a list of values:

```
values = ['foo', 'bar', 'baz']
other_vocabulary = SimpleVocabulary.fromValues(values)
```

Example 4 - registering as a reusable component, a named utility.

First a vocabulary factory needs to be created.

```
from zope.schema.interfaces import IVocabularyFactory
from zope.interface import provider

@provider(IVocabularyFactory)
def myvocabulary_factory(context):
    return myvocabulary
```

It has to be registered in ZCML as a utility:

```
<utility
  component=".vocab.myvocabulary_factory"
  name="example.myvocabulary"
/>
```

Retrieving a vocabulary

zope.schema's SimpleVocabulary objects are retrieved via factories registered as utilities.

To get one, use zope.component's getUtility:

```
from zope.component import getUtility
from zope.schema.interfaces import IVocabularyFactory

factory = getUtility(IVocabularyFactory, 'example.myvocabulary')
vocabulary = factory()
```

Getting a term

By term value

```
term = vocabulary.getTerm('value1')
value, token, term = (term.value, term.token, term.title)
```

Listing a vocabulary

Example - Iterate vocabulary SimpleTerm objects:

```
for term in vocabulary:
    print(term.value, term.token, term.title)
```

Dynamic vocabularies

Dynamic vocabularies' values may change run-time. They are usually generated based on some context data.

Note that the examples below need grok package installed and <grok:grok package="..."> directive in configure.zcml.

Complete example with portal_catalog query, vocabulary creation and form

```
"""
    A vocabulary example where vocabulary gets populated from portal_catalog query
    and then this vocabulary is used in Dexterity form.
"""

from five import grok
from plone.directives import form

from zope import schema
from z3c.form import button

from Products.CMFCore.interfaces import ISiteRoot, IFolderish
from Products.statusmessages.interfaces import IStatusMessage

from zope.schema.interfaces import IContextSourceBinder
from zope.schema.vocabulary import SimpleVocabulary, SimpleTerm

def make_terms(items):
    """ Create zope.schema terms for vocab from tuples """
    terms = [ SimpleTerm(value=pair[0], token=pair[0], title=pair[1]) for pair in
↪items ]
    return terms

@grok.provider(IContextSourceBinder)
def course_source(context):
    """
    Populate vocabulary with values from portal_catalog.

    @param context: z3c.form.Form context object (in our case site root)

    @return: SimpleVocabulary containing all areas as terms.
    """

    # Get site root from any content item using portal_url tool thru acquisition
    root = context.portal_url.getPortalObject()

    # Acquire portal catalog
    portal_catalog = root.portal_catalog

    # We need to get Plone site path relative to ZODB root
    # See traversing docs for more info about getPhysicalPath()
    site_physical_path = '/'.join(root.getPhysicalPath())

    # Target path we are querying
    folder_name = "courses"

    # Query all folder like objects in the target path
    # These portal_catalog query conditions are AND
    # but inside keyword query they are OR (the different content types
    # we are looking for)
    brains = portal_catalog.searchResults(path={ "query": site_physical_path + "/" +
↪folder_name },
                                         portal_type=["CourseInfo", "Folder"] )

    # Create a list of tuples (UID, Title) of results
```

```

result = [ (brain["UID"], brain["Title"]) for brain in brains ]

# Convert tuples to SimpleTerm objects
terms = make_terms(result)

return SimpleVocabulary(terms)

class IMyForm(form.Schema):
    """ Define form fields """

    name = schema.TextLine(
        title=u"Your name",
    )

    courses = schema.List(title=u"Promoted courses",
                          required=False,
                          value_type=schema.Choice(source=course_source)
    )

class MyForm(form.SchemaForm):
    """ Define Form handling

    This form can be accessed as http://yoursite/@@my-form

    """
    grok.name('my-form')
    grok.require('zope2.View')
    grok.context(ISiteRoot)

    schema = IMyForm
    ignoreContext = True

    @button.buttonAndHandler(u'Ok')
    def handleApply(self, action):
        data, errors = self.extractData()
        if errors:
            self.status = self.formErrorsMessage
            return

        # Do something with valid data here

        # Set status on this form page
        # (this status message is not bind to the session and does not go through_
↪ redirects)
        self.status = "Thank you very much!"

    @button.buttonAndHandler(u"Cancel")
    def handleCancel(self, action):
        """User cancelled. Redirect back to the front page.
        """

```

Complex example 2

```

from five import grok
from zope.schema.interfaces import IContextSourceBinder
from zope.schema.vocabulary import SimpleVocabulary, SimpleTerm
from Products.CMFCore.utils import getToolByName
from plone.i18n.normalizer import idnormalizer

```

```
def make_terms(items):
    """ Create zope.schema terms for vocab from tuples """
    terms = [ SimpleTerm(value=pair[0], token=pair[0], title=pair[1]) for pair in_
↪items ]
    return terms

@grok.provider(IContextSourceBinder)
def area_source(context):
    """
    Populate vocabulary with values from portal_catalog.

    Custom index name getArea contains utf-8 strings of
    possible area field values found on all content objects.

    @param context: Form context object.

    @return: SimpleVocabulary containing all areas as terms.
    """

    # Get catalog brain objects of all accommodation content
    accommodations = context.queryAllAccommodation()

    # Extract getArea index from the brains
    areas = [ a["getArea"] for a in accommodations ]
    # result will contain tuples (term, title) of acceptable items
    result = []

    # Create a form choice "do not filter"
    # which is always present
    result.append( ("all", _(u"All")) )

    # done list filter outs duplicates
    done = []
    for area in areas:
        if area != None and area not in done:

            # Archetype accessors return utf-8
            area_unicode = area.decode("utf-8")

            # Id must be 7-bit
            id = idnormalizer.normalize(area_unicode)
            # Decode area name to unicode
            # show that form shows international area
            # names correctly
            entry = (id, area_unicode)
            result.append(entry)
            done.append(area)

    # Convert tuples to SimpleTerm objects
    terms = make_terms(result)

    return SimpleVocabulary(terms)
```

For another example, see the Dynamic sources chapter in the Dexterity manual.

Registering a named vocabulary provider in ZCML

You can use `<utility>` in ZCML to register vocabularies by name and then refer them by name via `getUtility()` or in `zope.schema.Choice`.

```
<utility
  provides="zope.schema.interfaces.IVocabularyFactory"
  component="zope.app.gary.paths.Favorites"
  name="garys-favorite-path-references"
/>
```

Then you can refer to vocabulary by its name:

```
class ISearchCriteria(form.Schema):
    """ Alternative header flash animation/imagae """

    area = schema.Choice(source="garys-favorite-path-references", title=_("Area"),
↪required=False)
```

For more information see:

- [vocabularies API doc](#)
- [zope.component docs](#)

Forms, fields and widgets

Processing raw HTTP post requests

Description

How to read incoming HTTP POST values without form frameworks

Introduction

See *HTTP request object* for basics.

Here is an example view which checks if a form button has been pressed, and takes action accordingly.

Register the view in `configure.zcml`:

```
<configure
  xmlns="http://namespaces.zope.org/zope"
  xmlns:browser="http://namespaces.zope.org/browser"
  xmlns:plone="http://namespaces.plone.org/plone"
  i18n_domain="example.dexterityforms">

  ...

  <browser:page
    for="Products.CMFCore.interfaces.ISiteRoot"
    name="yourviewname"
    permission="zope2.View"
    class=".yourview.YourViewName"
```

```
        template="yourview.pt"
    />

</configure>
```

Code for the browser view, name the file `yourview.py`:

```
from Products.CMFCore.interfaces import ISiteRoot
from Products.Five.browser import BrowserView
from Products.statusmessages.interfaces import IStatusMessage

class YourViewName(BrowserView):

    def update(self):

        if "button-name" in self.request.form:

            messages = IStatusMessage(self.request)
            try:
                # do something
                messages.add("Button pressed")

            except Exception, e:
                logger.exception(e)
                messages.add(u"It did not work out. This exception came when_
↪processing the form:" + unicode(e))
```

Page template code, name the file `yourview.pt`:

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
    lang="en"
    metal:use-macro="here/main_template/macros/master"
    i18n:domain="ora.objects">
<body>
    <div metal:fill-slot="main">
        <tal:main-macro metal:define-macro="main">

            <h1 class="documentFirstHeading">
                Sample form
            </h1>

            <p class="documentDescription">
                Form description
            </p>

            <form action="@@yourviewname" method="POST"
                tal:define="foo view/update">
                <button type="submit" name="button-name">
                    Pres me
                </button>
            </form>

        </tal:main-macro>
    </div>
</body>
</html>
```

Magical Zope form variables

Zope provides some magical HTTP POST variable names which are automatically converted to native Python primitives by ZPublisher.

Quick explanation

If you have HTML:

```
<INPUT TYPE="text" NAME="member.age:int"></P><BR>
```

Then:

```
request.form["member.age"]
```

will return integer 30 instead of string "30".

Note: This behavior is hard-coded to ZPublisher and cannot be extended or disabled. The recommendation is not to use it, but do the conversion of data-types yourself or use a more high-level form framework like `z3c.form`.

More information

- <http://www.zope.org/Members/Zen/howto/FormVariableTypes>

z3c.form library

Description

`z3c.form` is flexible and powerful form library for Zope 3 applications. It is the recommended way to create complex Python-driven forms for Plone 4 and later versions.

Introduction

Plone uses `z3c.form` library with the following integration steps

- `plone.app.z3cform` provides Plone specific widgets and main template
- `plone.z3cform` integrates `z3c.form` with applications using Zope 2 mechanisms like acquisition
- `z3c.form` is a form library which can be used with any Python application using Zope 3 HTTP requests objects
- (Plone 4.4+ only) `plone.app.widgets` provide a better widget set over `z3c.form` default with more JavaScript-enabled features

Forms are modelled using *zope.schema* models written as Python classes. Widgets for modelled data are set by using either *plone.directives.form* hints set onto schema class or in `z3c.form.form.Form` based classes body.

Starting points to learn `z3c.form` in Plone

- Read about *creating schema-driven forms with Dexterity content subsystem*

- [Plone training manual](#)

Other related packages you might want to take a closer look

- Extra, more powerful widgets, from [collective.z3cform.widgets](#)
- Tabular data edit [collective.z3cform.datagridfield](#)
- Build JavaScript interfaces with [plone.app.jqueryui](#)
- Handling image and file fields with [plone.namedfile](#)
- Configuring forms with [plone.directives.form](#)

z3c.form big picture

The form model consists of:

self.request The incoming HTTP request.

self.context The Plone content item which was associated with the form view when URL traversing was done.

self.getContent() The actual object extracted from context and manipulated by the form if `ignoreContext` is not `False`.

self.status A message displayed at the top of the form to the user when the form is rendered. Usually it will be “Please correct the errors below”.

The call-chain for a form goes like this:

- `Form.update()` is called
 - [plone.autoform-based forms only] Calls `Form.updateFields()` - this will set widget factory methods for fields. If you want to customize the type of the widget associated with the field, do it here. If your form is not `plone.autoform`-based you need to edit `form.schema` widget factories on the module level code after the class has been constructed. The logic mapping widget hints to widgets is in `plone.autoform.utils`.
 - Calls `Form.updateWidgets()` - you can customize widgets at this point, if you override this method. The `self.widgets` instance is created based on the `self.fields` property.
 - Calls `Form.updateActions()`
 - * Calls the action handler (the handler for the button which was clicked)
 - * If it's an edit form, the action handler calls `applyChanges()` to store new values on the object and returns `True` if any value was changed.
- `Form.render()` is called
 - This renders the form as HTML, based on widgets and their templates.

Form

Simple boilerplate

Here is a minimal form implementation using `z3c.form` and `Dexterity`:

- Include `Dexterity` in your buildout as instructed by `Dexterity` manual
- Create Plone add-on product using [Paster](#)

Deprecated since version `may_2015`: Use [bobtemplates.plone](#)

- Register the form in `configure.zcml`:

```
<configure
  xmlns="http://namespaces.zope.org/zope"
  xmlns:browser="http://namespaces.zope.org/browser"
  xmlns:five="http://namespaces.zope.org/five"
  xmlns:genericsetup="http://namespaces.zope.org/genericsetup"
  xmlns:il8n="http://namespaces.zope.org/il8n"
  il8n_domain="example.dexterityforms">

  ...

  <browser:page
    for="Products.CMFCore.interfaces.ISiteRoot"
    name="my-form"
    permission="zope2.View"
    class=".form.MyForm"
    />

</configure>
```

- Toss `form.py` into your add-on product:

```
"""
    Simple sample form
"""

from plone.directives import form

from zope import schema
from z3c.form import button

from Products.CMFCore.interfaces import ISiteRoot
from Products.statusmessages.interfaces import IStatusMessage

class IMyForm(form.Schema):
    """ Define form fields """

    name = schema.TextLine(
        title=u"Your name",
    )

class MyForm(form.SchemaForm):
    """ Define Form handling

    This form can be accessed as http://yoursite/@my-form
    """

    schema = IMyForm
    ignoreContext = True

    label = u"What's your name?"
    description = u"Simple, sample form"

    @button.buttonAndHandler(u'Ok')
```

```
def handleApply(self, action):
    data, errors = self.extractData()
    if errors:
        self.status = self.formErrorsMessage
        return

    # Do something with valid data here

    # Set status on this form page
    # (this status message is not bind to the session and does not go thru_
↪redirects)
    self.status = "Thank you very much!"

@button.buttonAndHandler(u"Cancel")
def handleCancel(self, action):
    """User cancelled. Redirect back to the front page.
    """
```

Setting form status message

The form's global status message tells whether the form action succeeded or not.

The form status message will be rendered only on the form. If you want to set a message which will be visible even if the user renders another page after submitting the form, you need to use `Products.statusmessage`.

To set the form status message:

```
form.status = u"My message"
```

Emulating form HTTP POST in unit tests

- The HTTP request must include at least one buttons field.
- Form widget naming must match HTTP post values. Usually widgets have `form.widgets` prefix.
- You must emulate the ZPublisher behavior which automatically converts string input to Python primitives. For example, all choice/select values are Python lists.
- Some z3c widgets, like `<select>`, need to have `WIDGETNAME-empty-marker` value set to the integer 1 to be processed.
- Usually you can get the dummy HTTP request object via acquisition from `self.portal.REQUEST`

Example (incomplete):

```
layout = "accommodationsummary_view"

# Zope publisher uses Python list to mark <select> values
self.portal.REQUEST["form.widgets.area"] = [SAMPLE_AREA]
self.portal.REQUEST["form.buttons.search"] = u"Search"
view = self.portal.cards.restrictedTraverse(layout)

# Call update() for form
view.process_form()
print view.form.render()

# Always check form errors after update()
```

```
errors = view.errors
self.assertEqual(len(errors), 0, "Got errors:" + str(errors))
```

A more complete example:

```
# -*- coding: utf-8 -*-
from freitag.membership.testing import FREITAGMEMBERSHIP_INTEGRATION_TESTING
from z3c.form.interfaces import IFormLayer
from zope.interface import alsoProvides

import unittest

FORM_ID = 'password_reset'

class TestPasswordReset(unittest.TestCase):

    layer = FREITAGMEMBERSHIP_INTEGRATION_TESTING

    def setUp(self):
        self.portal = self.layer['portal']

    def test_nonexisting_fridge_rand(self):
        # create a password reset form
        self.portal.REQUEST["form.widgets.password"] = u'tatatata'
        self.portal.REQUEST["form.widgets.password_repeat"] = u'tatatata'
        self.portal.REQUEST["form.widgets.fridge_rand"] = 'nonexisting'
        self.portal.REQUEST["form.buttons.submit"] = u"Whatever"
        alsoProvides(self.portal.REQUEST, IFormLayer)
        form = self.portal.password_resetter.restrictedTraverse(FORM_ID)
        form.update()

        # data, errors = resetForm.extractData()
        data, errors = form.extractData()
        self.assertEqual(len(errors), 0)
```

Note that you will need to set `IFormLayer` on the request, to prevent a `ComponentLookupError`.

Changing form ACTION attribute

By default, the HTTP POST request is made to `context.absolute_url()`. However you might want to make the post go to an external server.

- See [how to set <form> action attribute](#)

Customizing form inner template

If you want to change the page template producing `<form>...</form>` part of the HTML code, follow the instructions below.

Note: Generally, when you have a template which extends Plone's `main_template` you need to use the `Products.Five.browser.pagetemplatefile.ViewPageTemplateFile` class.

Example:

```
# Do not mix with Products.Five.browser.pagetemplatefile.ViewPageTemplateFile
from zope.app.pagetemplate import ViewPageTemplateFile as Zope3PageTemplateFile

class AddHeaderAnimationForm(crud.AddForm):
    """ Present form for adding a header animation """

    template = Zope3PageTemplateFile("custom-form-template.pt")
```

Customizing form frame

Please see `plone.app.z3cform` README.

Rendering a form manually

You can directly create a form instance and call its `form.render()` method. This will output the full page HTML. However, there is a way to only render the form body payload.

First create a form and `update()`:

```
view.form = MyFormClass(self.context, self.request)
view.form.update()
```

Then you can invoke `plone.app.z3cform` macros directly to render the form body in your view's page template.

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
      xmlns:tal="http://xml.zope.org/namespaces/tal"
      xmlns:metal="http://xml.zope.org/namespaces/metal"
      xmlns:i18n="http://xml.zope.org/namespaces/i18n"
      metal:use-macro="here/main_template/macros/master"
      i18n:domain="plone.app.widgets"
      lang="en"
>
<body>

  <metal:main fill-slot="main">
    <tal:main-macro metal:define-macro="main">

      <h1 class="documentFirstHeading">Plone fields and widgets demo</h1>

      <div id="skel-contents">
        <tal:form repeat="form view/demos">

          <!-- plone.app.z3cform package provides view ploneform-macros
               which come with a helpers to render forms. This one
               will render the form body only. It also makes an assumption
               that form is presented in "view" TAL variable.

               -->
          <tal:with-form-as-view define="view nocall:form">
            <metal:block use-macro="form/@@ploneform-macros/titlelessform" />
          </tal:with-form-as-view>

        </tal:form>
      </div>
```



```

    </tal:main-macro>
  </metal:main>
</body>
</html>

```

Fields

A field is responsible for: 1) pre-populating form values from context 2) storing data to context after successful POST. Form fields are stored in the `form.fields` variable, which is an instance of the `Fields` class (ordered, dictionary-like).

Creating a field

Fields are created by adapting one or more `zope.schema` fields for `z3c.form` using the `Fields()` constructor.

Example of creating one field:

```

import zope.schema
import z3c.form.field

schema_field = zope.schema.TextLine()
form_fields = z3c.form.field.Fields(schema_field)

# This is a reference to newly created z3c.form.field.Field object
one_form_field = zfields.values()[0]

```

Another example:

```

import zope.schema
import z3c.form.field

...

field = zope.schema.Bool(
    __name__ = "death_autofill",
    title=(u"Fill missing timepoints"),
    description=(u"Automatically fill information in missing timepoints,
    ↪if they occur after the death time"),
    required=False,
    default=True)

# Construct z3c.form field
fields_objects = z3c.form.field.Fields(field)

# We can perform autofill only if we know the treatment time
form.fields += fields_objects

```

Adding a field to a form

Use the overridden `+=` operator of a `Fields` instance. Fields instances can be added to existing `Fields` instances.

Example:

```
self.form.fields += z3c.form.Fields(schema_field)
```

Modifying a field

Fields can be accessed by their name in `form.fields`. Example:

```
self.form.fields["myfieldname"].name = u"Foobar"
```

Accessing the schema of the field

A `zope.schema` Field is stored as a `field` attribute of a field. Example:

```
textline = self.form.fields["myfieldname"].field # zope.schema.TextLine
```

Note: There exist only one singleton instance of a schema during run-time. If you modify the schema fields, the changes are reflected to all subsequent form updates and other forms which use the same schema.

Read-only fields

There is `field.readonly` flag.

Example code:

```
class AREditForm(crud.EditForm):
    """ Form whose fields are dynamically constructed """

    def ar_editable(self):
        """ Arbitrary condition deciding whether fields on this form are
        patient=self.__parent__.__parent__
        if patient.getConfirmedAR() in (None, '', 'EDITABLE_AR'):
            return True
        return False

    @property
    def fields(self):
        """
        Dynamically create field data based on run-time constructed schema.

        Instead using static ``fields`` attribute, we use Python property
        which allows us to generate z3c.form.fields.Fields instance for the
        for run-time.
        """

        constructor = ARFormConstructor(self.context, self.context.context, self.
↪request)

        # Create z3c.form.field.Fields object instance
        fields = constructor.getFields()
```

```

if not self.ar_editable():
    # Disable all fields in edit mode if this form is locked out
    for f in fields.values():
        f.mode = z3c.form.interfaces.DISPLAY_MODE

return fields

```

You might also want to disable the *edit* button if none of the fields are editable:

```

# Make the edit button conditional
AREditSubForm.buttons["apply"].condition = lambda form: form.has_edit_button()

```

Note: You can also set `z3c.form.interfaces.DISPLAY_MODE` in `updateWidgets()` if you are not dynamically poking form fields themselves.

Warning: Do not modify fields on singleton instances (form or fields objects are shared between all forms). This causes problems on concurrent access.

Note: `zope.schema.Field` has a `readonly` property. `z3c.form.field.Field` does not have this property, but has the `mode` property. Do not confuse these two.

Dynamic schemas

Below is an example of how to include new schemas on the fly:

```

class EditForm(dexterity.EditForm, Helper):

    grok.context(IFlexibleContent)

    def updateFields(self):

        super(dexterity.EditForm, self).updateFields()
        sections = self.getSections()

        # See plone.app.z3cform.fieldsets.extensible for more examples
        for s in sections:

            # s = {'schema': <InterfaceClass your.app.content.flexiblecontent.
            # IBodyText>, 'id': u'title', 'name': u'Title'}
            if s == None:
                # This section has been removed from available flexi_blocks
                continue

            # convert zope schema interface to z3c.form.Fields instance
            schema = s["schema"]

            if not schema.providedBy(self.context):
                # We need to force the content item to provide
                # custom for interfaces or datamanager is not happy
                # Module z3c.form.datamanager, line 51, in adapted_context

```

```
        # TypeError: ('Could not adapt', <Item at /xxx/tydryd>,
        ↪<InterfaceClass xxx.app.content.flexiblecontent.IColumns>)
        alsoProvides(self.context, schema) # XXX: This is persistent change?

        # We need to manually apply hints from plone.directives.form, as
        # updateFields() does it for base schema earlier
        processFields(self, schema, permissionChecks=True)

    print "Final results"
    for name, field in self.fields.items():
        print str(name) + " " + str(field)
```

Date and time

Example:

```
class IDeal(form.Schema):
    """
    Deals and discounts item
    """

    validUntil = schema.Datetime(title=u"Valid until")
```

See

- <http://stackoverflow.com/questions/5776498/specify-datetime-format-on-zope-schema-date-on-plone>
- http://svn.zope.org/zope.schema/trunk/src/zope/schema/tests/test_datetime.py?rev=113055&view=auto

Making boolean field required

E.g. to make “Accept Terms and Conditions” checkbox

- <http://stackoverflow.com/questions/9670819/how-do-i-make-a-boolean-field-required-in-a-z3c-form>

Widgets

Widget are responsible for 1) rendering HTML code for input; 2) parsing HTTP post input.

Widgets are stored as the `widgets` attribute of a form. It is presented by an ordered dict-like `Widgets` class.

Widgets are only available after the form’s `update()` and `updateWidgets()` methods have been called. `updateWidgets()` will bind widgets to the form context. For example, vocabularies defined by name are resolved at this point.

A widget has two names:

- `widget.__name__` is the name of the corresponding field. Lookups from `form.widgets[]` can be done using this name.
- `widget.name` is the decorated name used in HTML code. It has the format `${form name}.${field set name}.${widget.__name__}`.

The Zope publisher will also mangle widget names based on what kind of input the widget takes. When an HTTP POST request comes in, Zope publisher automatically converts `<select>` dropdowns to lists and so on.

Setting a widget for a field

Using plone.directives.form schema hints

Example:

```
from plone.directives import form
from zope import schema
from plone.app.z3cform.wysiwyg import WysiwygFieldWidget

class ISampleSchema(form.Schema):

    # A fieldset with id 'extra' and label 'Extra information' containing
    # the 'footer' and 'dummy' fields. The label can be omitted if the
    # fieldset has already been defined.

    form.fieldset('extra',
                  label=u"Extra information",
                  fields=['footer', 'dummy']
                  )

    # Here a widget is specified as a dotted name.
    # The body field is also designated as the primary field for this schema

    form.widget(body='plone.app.z3cform.wysiwyg.WysiwygFieldWidget')
    form.primary('body')
    body = schema.Text(
        title=u"Body text",
        required=False,
        default=u"Body text goes here"
    )
```

More info

- Form schema hints

Setting widget for z3c.form plain forms

You can set field's widgetFactory after fields have been declared in form class body.

Example:

```
import zope.schema
import zope.interface

import z3c.form
from z3c.form.browser.checkbox import CheckBoxFieldWidget

class IReportSchema(zope.interface.Interface):
    """ Define reporter form fields """

    variables = zope.schema.List(
        title=u"Variables",
        description=u"Choose which variables to include in the output report",
        required=False,
        value_type=zope.schema.Choice(vocabulary="output_variables"))
```

```
class ReportForm(z3c.form.form.Form):
    """ A form to output a HTML report from chosen parameters """

    fields = z3c.form.field.Fields(IReportSchema)

    fields["variables"].widgetFactory = CheckBoxFieldWidget
```

Setting widget dynamically Form.updateWidgets()

Widget type can be set dynamically based on external conditions.

```
class EditForm9(EditForm):
    label = u'Rendering widgets as blocks instead of cells'

    grok.name('demo-collective.z3cform.datagrid-block-edit')

    def updateWidgets(self):
        super(EditForm9, self).updateWidgets()
        # Set a custom widget for a field for this form instance only
        self.fields['address'].widgetFactory = BlockDataGridFieldFactory
```

Accessing a widget

A widget can be accessed by its field's name. Example:

```
class MyForm(z3c.form.Form):

    def update(self):
        z3c.form.Form.update(self)
        widget = form.widgets["myfieldname"] # Get one widget

        for w in widget.items(): print w # Dump all widgets
```

Introspecting form widgets

Example:

```
from z3c.form import form

class MyForm(form.Form):

    def updateWidgets(self):
        """ Customize widget options before rendering the form. """
        form.Form.updateWidgets(self)

        # Dump out all widgets - note that each <fieldset> is a subform
        # and this function only concerns the current fieldset
        for i in self.widgets.items():
            print i
```

Reordering and hiding widgets

With Dexterity forms you can use `plone.directives.form`:

```
from z3c.form.interfaces import IAddForm, IEditForm

class IFlexibleContent(form.Schema):
    """
    Description of the Example Type
    """

    # -*- Your Zope schema definitions here ... -*-
    form.order_before(sections='title')
    form.mode(sections='hidden')
    form.mode(IEditForm, sections='input')
    form.mode(IAddForm, sections='input')
    sections = schema.TextLine(title=u"Sections")
```

Modifying a widget

Widgets are stored in the `form.widgets` dictionary, which maps *field name* to *widget*. The widget label can be different than the field name.

Example:

```
from z3c.form import form

class MyForm(form.Form):

    def updateWidgets(self):
        """ Customize widget options before rendering the form. """

        self.widgets["myfield"].label = u"Foobar"
```

If you want to have a completely different Python class for a widget, you need to override field's widget factory in the module body code after fields have been constructed in the class, or in the `update()` method for dynamically constructed fields:

```
def updateWidgets(self):
    self.fields["animation"].widgetFactory = HeaderFileFieldWidget
```

Reorder form widgets

`plone.z3cform` allows you to reorder the field widgets by overriding the `update` method of the form class.

Example:

```
from z3c.form import form
from plone.z3cform.fieldsets.utils import move

class MyForm(form.Form):

    def update(self):
        super(MyForm, self).update()
        move(self, 'fullname', before='*')
```

```
move(self, 'username', after='fullname')
super(ProfileRegistrationForm, self).update()
```

For more information about how to reorder fields see the `plone.z3cform` page at PyPI:

<<https://pypi.python.org/pypi/plone.z3cform#fieldsets-and-form-extenders>>‘_

Hiding fields

Here’s how to do it in pure `z3c.form`:

```
import z3c.form.interfaces
...

def updateWidgets(self):
    self.widgets["getAvailability"].mode = z3c.form.interfaces.HIDDEN_MODE
```

If you want to hide a widget that is part of a group, you cannot use the `updateWidgets` method. The groups and their widgets get initialized after the widgets have been updated. Before that, the `groups` variable is just a list of group factories. During the update method though, the groups have been initialized and have their own widget list each. For hiding widgets there, you have to access the group in the update method like so:

```
import z3c.form.interfaces
...

def update(self):
    for group in self.groups:
        if 'xxx' in group.widgets:
            group.widgets['xxx'].mode = z3c.form.interfaces.HIDDEN_MODE
```

`groups` itself is a list like object, you can also remove a complete group by removing it from the group dictionary.

Unprefixing widgets

By default each form widget gets a name prefixed by the form id. This allows you to combine several forms on the same page.

You can override this behavior in `updateWidgets()`:

```
# Remove prefix from form widget names, so that
# the names are actual names on the remote server
for widget in self.widgets.values():
    # form.widgets.foobar -> foobar
    widget.id = widget.name = widget.field.__name__
```

Note: Some templates, like `select_input.pt`, have hard-coded name suffixes like `:list` to satisfy ZPublisher machinery. If you need to get rid of these, you need to override the template.

Making widgets required conditionally

If you want to avoid hardwired `required` on fields and toggle then conditionally, you need to supply a dynamically modified schema field to the `z3c.form.field.Fields` instance of the form.

Example:

```
class ShippingAddressForm(CheckoutSubform):
    ignoreContext = True
    label = _(u"Shipping address")

    # Distinct fields on same <form> HTML element
    prefix = "shipping"

    def __init__(self, optional, content, request, parentForm):
        """
        @param optional: Whether shipping address should be validated or not.
        """
        subform.EditSubForm.__init__(self, content, request, parentForm)
        self.optional = optional

    @property
    def fields(self):
        """ Get the field definition for this form.

        Form class's fields attribute does not have to
        be fixed, it can be property also.
        """

        # Construct the Fields instance as we would
        # normally do in more static way
        fields = z3c.form.field.Fields(ICheckoutAddress)

        # We need to override the actual required from the
        # schema field which is a little tricky.
        # Schema fields are shared between instances
        # by default, so we need to create a copy of it
        if self.optional:
            for f in fields.values():
                # Create copy of a schema field
                # and force it unrequired
                schema_field = copy.copy(f.field) # shallow copy of an instance
                schema_field.required = False
                f.field = schema_field

        return fields
```

Setting widget types

By default, widgets for form fields are determined by FieldWidget adapters (defined in *ZCML*). You can override adapters per field using field's widgetFactory property.

Below is an example which creates a custom widget, its FieldWidget factory, and uses it for one field in one form:

```
from zope.component import adapter, getMultiAdapter
from zope.interface import implementer, implements, implementsOnly

from z3c.form.interfaces import IFieldWidget
from z3c.form.widget import FieldWidget

from plone.formwidget.namedfile.widget import NamedFileWidget, NamedImageWidget
```

```
class HeaderFileWidget(HeaderWidgetMixin, NamedFileWidget):

    # Get download url for HeaderAnimation object's file.
    # Download URL is set externally by edit sub form and
    download_url = None

class HeaderImageWidget(HeaderWidgetMixin, NamedImageWidget):
    pass

@implementer(IFieldWidget)
def HeaderFileFieldWidget(field, request):
    """ Factory for creating HeaderFileWidget which is bound to one field
    """
    return FieldWidget(field, HeaderFileWidget(request))

class EditHeaderAnimationSubForm(crud.EditSubForm):
    """
    """

    def updateWidgets(self):
        """ Enforce custom widget types which get file/image attachment URL right """
        # Custom widget types are provided by FieldWidget factories
        # before updateWidgets() is called
        self.fields["animation"].widgetFactory = HeaderFileFieldWidget

        crud.EditSubForm.updateWidgets(self)

        # Make edit form aware of correct image download URLs
        self.widgets["animation"].download_url = "http://mymagicalurl.com"
```

Alternatively, you can use `plone.directives.form` to add widget hints to form schema.

Widget save

After `form.update()` if the request was *save* and all data was valid, `form.applyChanges(data)` is called.

By default widgets use `datamanager.AttributeField` and try to store their values as a member attribute of the object returned by `form.getContent()`.

Widget value

The widget value, either from form POST or previous context data, is available as `widget.value` after the `form.update()` call.

Adding a CSS class

Widgets have a method `addClass()` to add extra CSS classes. This is useful if you have JavaScript/JQuery associated with your special form:

```
widget.addClass("myspecialwidgetclass")
```

Note that these classes are directly applied to `<input>`, `<select>`, etc. itself, and not to the wrapping `<div>` element.

Accessing the schema of the field

A `zope.schema.Field` is stored as a `field` attribute of a widget. Example:

```
textline = form.widgets["myfieldname"].field # zope.schema.TextLine
```

Warning: `Widget.field` is not a `z3c.form.field.Field` object.

Getting selection widget vocabulary value as human readable text

Example:

```
widget = self.widgets["myselectionlist"]

token = widget.value[0] # widget.value is list of unicode strings, each is token for
↳ the vocabulary

user_readable = widget.terms.getTermByToken(token).title
```

Example (page template)

```
<td tal:define="widget view/widgets/myselectionlist">
  <span tal:define="token python:widget.value[0]"
        tal:content="python:widget.terms.getTermByToken(token).title" />
</td>
```

Setting widget templates

You might want to customize the template of a widget to have custom HTML code for a specific use case.

Setting the template of an individual widget

First copy the existing page template code of the widget. For basic widgets you can find the template in the `z3c.form` source tree.

`yourwidget.pt` (text area widget copied over an example text)

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:tal="http://xml.zope.org/namespaces/tal"
      tal:omit-tag="">

<!-- Sections widget custom templates -->

<textarea
  id="" name="" class="" cols="" rows=""
  tabindex="" disabled="" readonly="" accesskey=""
  tal:attributes="id view/id;
                  name view/name;
                  class view/klass;
                  style view/style;
                  title view/title;
```

```
lang view/lang;
onclick view/onclick;
ondblclick view/ondblclick;
onmousedown view/onmousedown;
onmouseup view/onmouseup;
onmouseover view/onmouseover;
onmousemove view/onmousemove;
onmouseout view/onmouseout;
onkeypress view/onkeypress;
onkeydown view/onkeydown;
onkeyup view/onkeyup;
disabled view/disabled;
tabindex view/tabindex;
onfocus view/onfocus;
onblur view/onblur;
onchange view/onchange;
cols view/cols;
rows view/rows;
readonly view/readonly;
accesskey view/accesskey;
onselect view/onselect"
tal:content="view/value" />
</html>
```

Now you can override the template factory in the `updateWidgets()` method of your form class

```
from zope.browserpage.viewpagetemplatefile import ViewPageTemplateFile as _
↳ Z3ViewPageTemplateFile
from z3c.form.interfaces import INPUT_MODE

class AddForm(DefaultAddForm):

    def updateWidgets(self, prefix=None):
        """ """
        # Call parent to set-up initial widget data
        DefaultAddForm.updateWidgets(self, prefix=prefix)

        # Note we need to be discreet to different form modes (view, edit, hidden)
        if self.fields["sections"].mode == INPUT_MODE:

            # Modify a widget with certain name for our purposes
            widget = self.widgets["sections"]

            # widget.template is a template factory -
            # Widget.render() will associate later this factory with the widget
            widget.template = Z3ViewPageTemplateFile("templates/sections.pt")
```

You can also interact with your form class instance from the widget template

```
<!-- Some hidden JSON data for our Javascripts by calling a method on our form class -
↳ ->
<span style="display:none" tal:content="view/form/getBlockPlanJSON" />
```

Setting template for your own widget type

You can set the template used by the widget with the `<z3c:widgetTemplate>` ZCML directive

```
<z3c:widgetTemplate
    mode="display"
    widget=".interfaces.INamedFileWidget"
    layer="z3c.form.interfaces.IFormLayer"
    template="file_display.pt"
/>
```

You can also enforce the widget template in the `render()` method of the widget class:

```
from zope.component import adapter, getMultiAdapter
from zope.interface import implementer, implements, implementsOnly
from zope.app.pagetemplate.viewpagetemplatefile import ViewPageTemplateFile

from z3c.form.interfaces import IFieldWidget, INPUT_MODE, DISPLAY_MODE, HIDDEN_MODE
from z3c.form.widget import FieldWidget

from plone.formwidget.namedfile.widget import NamedFileWidget, NamedImageWidget

class HeaderFileWidget(NamedFileWidget):
    """ Subclass widget to use a custom template """

    display_template = ViewPageTemplateFile("header_file_display.pt")

    def render(self):
        """See z3c.form.interfaces.IWidget."""

        if self.mode == DISPLAY_MODE:
            # Enforce template and do not query it from the widget template factory
            template = self.display_template

        return NamedFileWidget.render(self)
```

Widget template example:

```
<span id="" class="" i18n:domain="plone.formwidget.namedfile"
    tal:attributes="id view/id;
        class view/klass;
        style view/style;
        title view/title;
        lang view/lang;
        onclick view/onclick;
        ondblclick view/onclick;
        onmousedown view/onmousedown;
        onmouseup view/onmouseup;
        onmouseover view/onmouseover;
        onmousemove view/onmousemove;
        onmouseout view/onmouseout;
        onkeypress view/onkeypress;
        onkeydown view/onkeydown;
        onkeyup view/onkeyup"
    tal:define="value view/value;
        exists python:value is not None">
    <span tal:define="fieldname view/field/___name___ | nothing;
        filename view/filename;
        filename_encoded view/filename_encoded;"
        tal:condition="python: exists and fieldname">
    <a tal:content="filename"
        tal:attributes="href string:${view/download_url}">Filename</a>
```

```
<span class="discreet"> &mdash; <span tal:define="sizekb view/file_size" _
↪tal:replace="sizekb">100</span> KB</span>
</span>
<span tal:condition="not:exists" class="discreet" i18n:translate="no_file">
    No file
</span>
</span>
```

Setting widget frame template

You can change how the frame around each widget is rendered in the widget rendering loop. This frame has elements like label, required marker, field description and so on.

For instructions see [plone.app.z3cform README](#)

Combined widgets

You can combine multiple widgets to one with `z3c.form.browser.multil.MultiWidget` and `z3c.form.browser.object.ObjectWidget` classes.

Example how to create a min max input widget.

Python code to setup the widget:

```
import zope.interface
import zope.schema
from zope.schema.fieldproperty import FieldProperty

import z3c.form
from z3c.form.object import registerFactoryAdapter

class IMinMax(zope.interface.Interface):
    """ Helper schema for min and max fields """

    min = zope.schema.Float(required=False)

    max = zope.schema.Float(required=False)

@zope.interface.implementer(IMinMax)
class MinMax(object):
    """ Store min-max field values """
    min = FieldProperty(IMinMax['min'])
    max = FieldProperty(IMinMax['max'])

registerFactoryAdapter(IMinMax, MinMax)

....

field = zope.schema.Object(__name__='mixmax', title=label, schema=IMinMax, _
↪required=False)
```

Then we do some widget marking in `updateWidgets()`:

```
def updateWidgets(self):
    """
    """

    super(FilteringGroup, self).updateWidgets()

    # Add min and max CSS class rendering hints
    for widget in self.widgets.values():
        if isinstance(widget, z3c.form.browser.object.ObjectWidget):
            widget.template = Z3ViewPageTemplateFile("templates/minmax.pt")
            widget.addClass("min-max-widget")
            zope.interface.alsoProvides(widget, IFilterWidget)
```

And then the page template which renders both 0. widget (min) and 1. widget (max) on the same line.

```
<div class="min-max-widget"
    tal:define="widget0 python:view.subform.widgets.values()[0]; widget1 python:view.
    ↪subform.widgets.values()[1];">

    <tal:comment>
        <!-- Use label from the first widget -->
    </tal:comment>

    <div class="label">
        <label tal:attributes="for widget0/id">
            <span il8n:translate=""
                tal:content="widget0/label">label</span>
        </label>
    </div>

    <div class="widget-left" tal:define="widget widget0">

        <div tal:content="structure widget/render">
            <input type="text" size="24" value="" />
        </div>

    </div>

    <div class="widget-separator">
        -
    </div>

    <div class="widget-right" tal:define="widget widget1">

        <div class="widget" tal:content="structure widget/render">
            <input type="text" size="24" value="" />
        </div>

    </div>

    <div tal:condition="widget0/error"
        tal:replace="structure widget/error/render">error</div>

    <div class="error" tal:condition="widget1/error"
        tal:replace="structure widget1/error/render">error</div>
```

```
<div style="clear: both"><!-- --></div>

<input name="field-empty-marker" type="hidden" value="1"
      tal:attributes="name string:${view/name}-empty-marker" />

</div>
```

Buttons

Buttons enable actions in forms. `AddForm` and `EditForm` base classes come with default buttons (*Save*).

More information in `z3c.form` documentation

- <http://packages.python.org/z3c.form/button.html>

Adding a button to form

The easiest way to add handlers for buttons is to use a function decorator `z3c.form.button.buttonAndHandler()`.

The first parameter is the user visible label and the second one is the `<input>` name.

Example:

```
from z3c.form import button

class Form(...):

    @button.buttonAndHandler(_('Add'), name='add')
    def handle_add(self, action):
        data, errors = self.extractData()
        if errors:
            self.status = "Please correct errors"
            return

        self.applyChanges(data)
        self.status = _(u"Item added successfully.")
```

The default `z3c.form.form.AddForm` and `z3c.form.form.EditForm` *Add* and *Save* button handler calls are good code examples.

- <https://github.com/zopefoundation/z3c.form/blob/master/src/z3c/form/form.py>

If you created a form based on another form, the buttons defined on that other form get lost. To prevent that, you must explicitly add the buttons of the base class in your form class:

```
from z3c.form import button
from z3c.form.form import EditForm

class Form(EditForm):

    buttons = EditForm.buttons.copy()

    @button.buttonAndHandler(...)
    def handle_add(...):
        ...
```


Adding buttons conditionally

The `buttonAndHandler` decorator can accept a condition argument. The condition should be a function that accepts the form as an argument and returns a boolean. Example, a button that only shows when a condition is met:

```
@button.buttonAndHandler(
    u"Delete Event",
    name="handleDelete",
    condition=lambda form: form.okToDelete()
)
def handleDelete(self, action):
    """
    Delete this event.
    """
    ...

    self.status = "Event deleted."
```

Manipulating form buttons programmatically

You want to manipulate buttons if you want to hide buttons dynamically, manipulate labels, etc.

Buttons are stored in `buttons` class attribute.

Warning: Button storage is shared between all form instances, do not mutate its content. Instead create a copy of it if you wish to have form-specific changes.

Reading buttons

Example:

```
self.mobile_form_instance = MobileForm(self.context, self.request)

for i in self.mobile_form_instance.buttons.items(): print i
('apply', <Button 'apply' u'Apply'>)
```

Removing or hiding buttons

Here is an example how to hide all buttons from a certain form instance.

Example:

```
import copy

def update(self):
    # Hide form buttons

    # Create immutable copy which you can manipulate
```

```
self.mobile_form_instance.buttons = copy.deepcopy(self.mobile_form_instance.
↳ buttons)

# Remove button using dictionary style delete
for button_id in self.mobile_form_instance.buttons.keys():
    del self.mobile_form_instance.buttons[button_id]
```

Adding buttons dynamically

In the example below, the Buttons array is already constructed dynamically and we can manipulate it:

```
def setActions(self):
    """ Add button to the form based on dynamic conditions. """

    if self.isSaveEnabled():

        but = button.Button("save", title=u"Save")
        self.form.buttons += button.Buttons(but)

        self.form.buttons._data_keys.reverse() # Fix Save button to left

        handler = button.Handler(but, self.form.__class__.handleSave)
        self.form.handlers.addHandler(but, handler)
```

Subforms

Subforms are embedded z3c forms inside a master form.

Subforms may have their own buttons or use the controls from the master form. You need to call `update()` manually for subforms.

More info

- <http://packages.python.org/z3c.form/subform.html>

Adding an action to parent and subform

Parent and subform actions must be linked.

Example:

```
class CheckoutForm(z3c.form.form.EditForm):

    @button.buttonAndHandler(_('Continue'), name='continue')
    def handleContinue(self, action):
        """ Extract the checkout data to session and redirect to payment Arbitrary_
↳ checkout screen.

        Note:

        """

        # Following has been copied from z3c.form.form.EditForm
```

```

        data, errors = self.extractData()
        if errors:
            self.status = self.formErrorsMessage
            return

        changes = self.applyChanges(data)

        if changes:
            self.status = self.successMessage
        else:
            self.status = self.noChangesMessage

class CheckoutSubform(subform.EditSubForm):
    """ Add support for continue action. """

    def execute(self):
        """
        Make sure that the form is refreshed when parent
        form Continue is pressed.
        """

        data, errors = self.extractData()
        if errors:
            self.errors = errors
            self.status = self.formErrorsMessage
            return errors

        content = self.getContent()
        z3c.form.form.applyChanges(self, content, data)

        return None

    @button.handler(CheckoutForm.buttons['continue'])
    def handleContinue(self, action):
        """ What happens when the parent form button is pressed """
        self.execute()

```

Creating subforms at run-time

Below is an example how to convert existing form instance to be used as a subform in another form:

```

def convertToSubForm(self, form_instance):
    """
    Make existing form object behave like subform object.

    * Do not render <form> frame

    * Do not render actions

    @param form_instance: Constructed z3c.form.form.Form object
    """

    # Create mutable copy which you can manipulate
    form_instance.buttons = copy.deepcopy(form_instance.buttons)

```

```
# Remove subform action buttons using dictionary style delete
for button_id in form_instance.buttons.keys():
    del form_instance.buttons[button_id]

if HAS_WRAPPER_FORM:
    # Plone 4 / Plone 3 compatibility
    zope.interface.alsoProvides(form_instance, IWrappedForm)

# Use subform template - this prevents getting embedded <form>
# elements inside the master <form>
import plone.z3cform
#from zope.pagetemplatefile import ViewPageTemplateFile as Zope3PageTemplateFile
from zope.app.pagetemplate import ViewPageTemplateFile as Zope3PageTemplateFile
from zope.app.pagetemplate.viewpagetemplatefile import BoundPageTemplate
template = Zope3PageTemplateFile('subform.pt', os.path.join(os.path.dirname(plone.
→z3cform.__file__), "templates"))
form_instance.template = BoundPageTemplate(template, form_instance)
```

Note: If possible, try to construct your form class hierarchy so that you can use the same class mix-in for normal forms and subforms.

CRUD form

CRUD (Create, read, update, delete) forms manage list of objects.

CRUD form elements:

- Add form creates new objects and renders the form below the table
- Edit sub-form edits existing object and renders one table row
- Edit form lists all objects and allows deleting them (table master)
- CRUD form orchestrates the whole thing and renders add and edit forms
- `view_schema` outputs read-only fields in CRUD table
- `update_schema` outputs editable fields in CRUD table. Usually you want either `view_schema` or `update_schema`.
- `add_schema` outputs add form.

Note: the `context` attribute of add and edit form is the parent CRUD form. The `context` attribute of an edit subform is the edit form.

Examples

- <https://pypi.python.org/pypi/plone.z3cform#crud-create-read-update-and-delete-forms>

Displaying the status message in a non-standard location

By default, the status message is rendered inside `plone.app.z3cform.macros.pt` above the form:

```
<metal:define define-macro="titlelessform">

  <tal:status define="status view/status" condition="status">
    <dl class="portalMessage error" tal:condition="view/widgets/errors">
      <dt i18n:domain="plone" i18n:translate="">
        Error
      </dt>
      <dd tal:content="status" />
    </dl>
    <dl class="portalMessage info" tal:condition="not: view/widgets/errors">
      <dt i18n:domain="plone" i18n:translate="">
        Info
      </dt>
      <dd tal:content="status" />
    </dl>
  </tal:status>
```

We can decouple the status message from the form, without overriding all the templates, by copying status message variable to another variable and then playing around with it in our wrapper view template.

Form class:

```
class HolidayServiceSearchForm(form.Form):
    """
    """

    @button.buttonAndHandler(_(u"Search"))
    def searchHandler(self, action):
        """ Search form submit handler for product card search.
        """

        data, errors = self.extractData()
        if len(self.search_results) == 0:
            self.status = _(u"No holiday services found.")
        else:
            msgid = _("found_results", default=u"Found ${results} holiday services.",
↪mapping={u"results" : len(self.search_results)})
            self.status = self.context.translate(msgid)

        ...

        # Use non-standard location to display the status
        # for success messages
        if len(self.widgets.errors) == 0:
            self.result_message = self.status
            self.status = None

class HolidayServiceSearchView(FormWrapper):
    """ HolidayService browser view
    """

    form = HolidayServiceSearchForm

    def result_message(self):
        """ Display result message in non-standard location """

        if len(self.form_instance.widgets.errors) == 0:
            # Do not display form highlight errors here
```

```
return self.form_instance.result_message
```

... and then we can use a special `result_message` view accessor in our view template code

```
<tal:comment replace="nothing">Form submit anchor</tal:comment>
<a name="searched" />

<tal:status define="status view/result_message" condition="python:status != None">
  <dl class="portalMessage info">
    <dt i18n:domain="plone" i18n:translate="">
      Info
    </dt>
    <dd tal:content="status" />
  </dl>
</tal:status>
```

Storage format and data managers

By default, `z3c.form` reads incoming context values as the object attributes. This behavior can be customized using data managers.

You can, for example, use Python dictionaries to read and store form data.

- <http://packages.python.org/z3c.form/datamanager.html>

Custom content objects

The following hack can be used if you have an object which does not conform your form interface and you want to expose only certain object attribute to the form to be edited.

Example:

```
class ISettings(zope.interface.Interface):

    # This maps to Archetypes field confirmedAR on SitsPatient
    confirmedAR = zope.schema.Choice(
        title=(u"Confirm adhere reactions"),
        description=(u"Confirm that all adhere reactions regarding the patient_
↪ life cycle have been entered here and there will be no longer adhere reaction data
↪"),
        vocabulary=make_zope_schema_vocabulary(ADVERSE_STATUS_VOCABULARY))

class ARSettingsForm(form.Form):
    """ General settings for all adhere reactions """

    fields = Fields(ISettings)

    def getContent(self):
        """ """

        # Create a temporary object holding the settings values out of the patient

        class TemporarySettingsContext(object):
            zope.interface.implements(ISettings)

        obj = TemporarySettingsContext()
```

```

        # Copy values we want to expose to the form from Plone context item to the_
        ↪temporary object
        obj.confirmedAR = self.context.confirmedAR

    return obj

```

Note: Since `getContent()` is also used in `applyChanges()`, you need to override `applyChanges()` as well to save values correctly to a persistent object.

Custom change applying

The default, the behavior of the `z3c.form` edit form is to write incoming data as the attributes of the object returned by `getContent()`.

You can override this behavior by overriding `applyChanges()` method.

Example:

```

def applyChanges(self, data):
    """
    Reflect confirmed status to Archetypes schema.

    @param data: Dictionary of cleaned form data, keyed by field
    """

    # This is the context given to the form when the form object was constructed
    patient = self.context

    assert ISitsPatient.providedBy(patient) # safety check

    # Call archetypes field mutator to store the value on the patient object
    patient.setConfirmedAR(data["confirmedAR"])

```

WYSIWYG widgets

By using `plone.directives.form` and `plone.app.z3cform` packages you can do:

```

from plone.app.z3cform.wysiwyg import WysiwygFieldWidget

from mfabrik.plonezohointegration import _

class ISettings(form.Schema):
    """ Define schema for settings of the add-on product """

    form.widget(contact_form_prefix=WysiwygFieldWidget)
    contact_form_prefix = schema.Text(
        title=_(u"Contact form top text"),
        description=_(u"Custom text for the long contact form upper part"),
        required=False,
        default=u"")

```

More information

- <https://pypi.python.org/pypi/plone.directives.form>

Wrapped and non-wrapped forms

A `z3c.form.form.Form` object is “wrapped” when it is rendered inside Plone page frame and having acquisition chain in intact.

Since `plone.app.z3cform 0.5.0` the behavior goes like this:

- Plone 3 forms are automatically wrapped
- Plone 4 forms are unwrapped

The wrapper is a `plone.z3cform.interfaces.IWrappedForm` *marker interface* on the form object, applied it after the form instance has been constructed. If this marker interface is not applied, `plone.z3cform.ZopeTwoFormTemplateFactory` tries to embed the form into Plone page frame. If the form is not intended to be rendered as a full page form, this usually leads to the following exception:

```
*** ContentProviderLookupError: plone.htmlhead
```

The form tries to render the full Plone page. Rendering this page needs an acquisition chain set-up for the view and the template. Embedded forms do not have this, or it would lead to recursion error.

If you are constructing form instances manually and want to render them without Plone page decoration, you must make sure that automatic form wrapping does not take place:

```
import zope.interface
from plone.z3cform.interfaces import IWrappedForm

class SomeView(BrowserView):

    def init(self):
        """ Constructor embedded sub forms """

        # Construct few embedded forms
        self.mobile_form_instance = MobileForm(
            self.context, self.request)
        zope.interface.alsoProvides(
            self.mobile_form_instance, IWrappedForm)

        self.publishing_form_instance = PublishingForm(
            self.context, self.request)
        zope.interface.alsoProvides(
            self.publishing_form_instance, IWrappedForm)

        self.override_form_instance = getMultiAdapter(
            (self.context, self.request),
            IOverrideForm)
        zope.interface.alsoProvides(
            self.override_form_instance, IWrappedForm)
```

Embedding z3c.form forms in portlets, viewlets and views

By default, when `plone.app.z3cform` is installed through the add-on installer, all forms have full Plone page frame. If you are rendering forms inside non-full-page objects, you need to change the default template.

Below is an example how to include a `z3c.form`-based form in a portlet.

Note: `plone.app.z3cform` version 0.5.1 or later is needed, as older versions do not support overriding `form.action` property.

You need the following:

- a `z3c.form` class
- the viewlet/portlet class
- A form wrapper template which renders the frame around the form. The default version renders the whole Plone page frame — you don't want this when the form is embedded, otherwise you get infinite recursion (plone page having a form having a plone page...)
- Portlet/viewlet template which refers to the form
- ZCML to register all components

Portlet code:

```
from plone.z3cform.layout import FormWrapper

class PortletFormView(FormWrapper):
    """ Form view which renders z3c.forms embedded in a portlet.

    Subclass FormWrapper so that we can use custom frame template. """

    index = ViewPageTemplateFile("formwrapper.pt")

class Renderer(base.Renderer):
    """ z3c.form portlet renderer.

    Instantiate form and wrap it to a special layout template
    which will give the form suitable frame to be used in the portlet.

    We also set a form action attribute, so that
    the browser goes to another page after the form has been submitted
    (we really don't know what kind of page the portlet is displayed
    and is it safe to submit forms there, so we do this to make sure).
    The action page points to a browser:page view where the same
    form is displayed as full-page form, giving the user to better
    user experience to fix validation errors.
    """

    render = ViewPageTemplateFile('zohocrmcontact.pt')

    def __init__(self, context, request, view, manager, data):
        base.Renderer.__init__(self, context, request, view, manager, data)
        self.form_wrapper = self.createForm()

    def createForm(self):
        """ Create a form instance.

        @return: z3c.form wrapped for Plone 3 view
        """

        context = self.context.aq_inner
```

```
returnURL = self.context.absolute_url()

# Create a compact version of the contact form
# (not all fields visible)
form = ZohoContactForm(context, self.request, returnUrlHint=returnURL,
↪full=False)

# Wrap a form in Plone view
view = PortletFormView(context, self.request)
view = view.__of__(context) # Make sure acquisition chain is respected
view.form_instance = form

return view

def getContactFormURL(self):
    """ For rendering the form link at the bottom of the portlet.

    @return: URL leading to the full contact form
    """
    return self.form_wrapper.form_instance.action
```

formwrapper.pt is just a dummy form view template which wraps the form. This differs from standard form wrapper by *not* rendering Plone main layout around the form.

```
<div class="portlet-form">
  <div tal:replace="structure view/contents" />
</div>
```

Then the portlet template itself (zohoportlet.pt) renders the portlet. The form is rendered using: <form tal:replace="structure view/form_wrapper" />.

```
<dl class="portlet portletZohoCRMContact"
    id:n:domain="mfabrik.plonezohointegration">

  <dt class="portletHeader">
    <span class="portletTopLeft"></span>
    <span id:n:translate="portlet_title">
      Contact Us
    </span>
    <span class="portletTopRight"></span>
  </dt>

  <dd class="portletItem odd">
    <form tal:replace="structure view/form_wrapper" />
  </dd>

  <dd class="portletFooter">
    <span class="portletBottomLeft"></span>
    <a href=""
      tal:attributes="href view/getContactFormURL"
      id:n:translate="box_more_news_link">
      Longer contact form&hellip;
    </a>
    <span class="portletBottomRight"></span>
  </dd>

</dl>
```

Note: Viewlets behave a little differently, since they do some acquisition chain mangling when you assign variables to `self`. Thus you should never have `self.view = view` or `self.form = form` in a viewlet.

Template example for viewlet (don't do `self.form_wrapper`)

```
<div id="my-viewlet">
    <form tal:replace="structure python:view.createForm() ()" />
</div>
```

Then the necessary parts of form itself:

```
class IZohoContactForm(zope.interface.Interface):
    """ Form field definitions for Zoho contact forms """

    first_name = schema.TextLine(title=_(u"First name"))

    last_name = schema.TextLine(title=_(u"Last name"))

    company = schema.TextLine(title=_(u"Company / organization"), description=_(u"The
    ↪organization which you represent"))

    email = schema.TextLine(title=_(u"Email address"), description=_(u"Email address
    ↪we will use to contact you"))

    phone_number = schema.TextLine(title=_(u"Phone number"),
    ↪description=_(u"Your phone number in international
    ↪format. E.g. +44 12 123 1234"),
    ↪required=False,
    ↪default=u"")

    returnURL = schema.TextLine(title=_(u"Return URL"),
    ↪description=_(u"Where the user is taken after the
    ↪form is successfully submitted"),
    ↪required=False,
    ↪default=u"")

class ZohoContactForm(Form):
    """ z3c.form used to handle the new lead submission.

    This form can be rendered

    * standalone (@@zoho-contact-form view)

    * embedded into the portlet

    ..note::

        It is recommended to use a CSS rule
        to hide form descriptions when rendered in the portlet to save
        some screen estate.

    Example CSS::

        .portletZohoCRMContact .formHelp {
            display: none;
        }
    """
```

```

"""

fields = Fields(IZohoContactForm)

label = _(u"Contact Us")

description = _(u"If you are interested our services leave your contact_
↪information below and our sales representatives will contact you.")

ignoreContext = True

def __init__(self, context, request, returnUrlHint=None, full=True):
    """
    @param returnUrlHint: Should we enforce return URL for this form

    @param full: Show all available fields or just required ones.
    """
    Form.__init__(self, context, request)
    self.all_fields = full

    self.returnURLHint = returnUrlHint

@property
def action(self):
    """ Rewrite HTTP POST action.

    If the form is rendered embedded on the others pages we
    make sure the form is posted through the same view always,
    instead of making HTTP POST to the page where the form was rendered.
    """
    return self.context.portal_url() + "/@@zoho-contact-form"

def updateWidgets(self):
    """ Make sure that return URL is not visible to the user.
    """
    Form.updateWidgets(self)

    # Use the return URL suggested by the creator of this form
    # (if not acting standalone)
    self.widgets["returnURL"].mode = z3c.form.interfaces.HIDDEN_MODE
    if self.returnURLHint:
        self.widgets["returnURL"].value = self.returnURLHint

    # Prepare compact version of this formw
    if not self.all_fields:
        # Hide fields which we don't want to bother user with
        self.widgets["phone_number"].mode = z3c.form.interfaces.HIDDEN_MODE

@button.buttonAndHandler(_('Send contact request'), name='ok')
def send(self, action):
    """ Form button handler. """

    data, errors = self.extractData()

    if not errors:

```

```

settings = self.getZohoSettings()
if settings is None:
    self.status = _(u"Zoho is not configured in Site Setup. Please_
↳contact the site administration.")
    return

crm = CRM(settings.username, settings.password, settings.apikey)

# Fill in data going to Zoho CRM
lead = {
    "First Name" : data["first_name"],
    "Last Name" : data["last_name"],
    "Company" : data["company"],
    "Email" : data["email"],
}

phone = data.get("phone_number", "")
if phone != "":
    # Only pass phone number to Zoho if it's set
    lead["Phone"] = phone

# Pass in all prefilled lead fields configured in the site setup
lead.update(self.parseExtraFields(settings.crm_lead_extra_data))

# Open Zoho API connection
try:
    # This will raise ZohoException and nuke the request
    # if Zoho credentials are wrong
    crm.open()

    # Make sure that wfTrigger is true
    # and Zoho does workflow actions for the new leads
    # (like informing sales about the availability of the lead)
    crm.insert_records([lead], {"wfTrigger" : "true"})
except IOError:
    # Network down?
    self.status = _(u"Cannot connect to Zoho servers. Please contact web_
↳site administration")
    return

ok_message = _(u"Thank you for contacting us. Our sales representatives_
↳will come back to you in few days")

# Check whether this form was submitted from another page
returnURL = data.get("returnURL", "")

if returnURL != "" and returnURL is not None:

    # Go to page where we were sent and
    # pass the confirmation message as status message (in session)
    # as we are not in the control of the destination page
    from Products.statusmessages.interfaces import IStatusMessage
    messages = IStatusMessage(self.request)
    messages.addStatusMessage(ok_message, type="info")
    self.request.response.redirect(returnURL)
else:
    # Act standalone

```

```
        self.status = ok_message
    else:
        # errors on the form
        self.status = _(u>Please fill in all the fields")
```

Further reading

This example code was taken from the `mfabrik.plonezohointegration` product which is in the Plone collective.

Validators

Introduction

There are three kind of validation hooks you can use with `z3c.form`

- `zope.schema` field parameter specific
- `zope.schema` `@invariant` (validation is model specific)
- `zope.schema` constraint (validation is model specific)
- `z3c.form` (validation is bound to the form instance)

Field specific internal validators

When you define your field with `zope.schema` you can enable flags for field internal validation. This include e.g.

- `required` is field required on the form or not
- `min` and `max` for number based fields

Example:

```
class LocalizationOfStenosisForm(form.Schema):

    degreeOfStenosis = schema.Float(
        title=u'Degree of stenosis %',
        required=False,
        min=0.0,
        max=100.0
    )
```

For available internal validation options, see the field source code in `zope.schema` package.

Constraint validators

`zope.schema` fields take a callable argument `constraint` which defines a Python function validating the incoming value.

```
import zope.interface

def lastNameConstraint(value):
    if value and value == value.lower():
```

```

        raise zope.interface.Invalid(u"Name must have at least one capital letter")
    return True

class IPerson(zope.interface.Interface):

    lastName = zope.schema.TextLine(
        title=u'Last Name',
        description=u'The person's last name.',
        default=u'',
        required=True,
        constraint=lastNameConstraint)

```

For more information, see `zope.schema` documentation.

Invariant validators

Invariants validator do validations between fields. They are checked after the single field validations are processed.

Example: With invariants it is possible to check if start date is before end date:

```

from zope.interface import Invalid
from zope.interface import invariant

@provider(IFormFieldProvider)
class ISomeDates(form.Schema):

    @invariant
    def start_before_end(data):
        if data.start > data.end:
            raise Invalid(_(u'Start must be before end!'))

```

Form widget validators

Example: How to use widget specific validators with `z3c.form`:

```

from z3c.form import validator
import zope.component

class IZohoContactForm(form.Schema):
    """ Form field definitions for Zoho contact forms """

    phone_number = schema.TextLine(
        title=_(u'Phone number'),
        description=_(u'Your phone number in international format. E.g. +44 12 123_
↪1234'),
        required=False,
        default=u''
    )

class PhoneNumberValidator(validator.SimpleFieldValidator):
    """ z3c.form validator class for international phone numbers """

    def validate(self, value):
        """ Validate international phone number on input """
        allowed_characters = '+- () / 0123456789'

```

```
    if value is None:
        return

    value = value.strip()

    if not value:
        # Assume empty string = no input
        return

    # The value is not required
    for ch in value:
        if ch not in allowed_characters:
            raise zope.interface.Invalid(
                _(u'Phone number contains bad characters')
            )

    if len(value) < 7:
        raise zope.interface.Invalid(_(u'Phone number is too short'))

# Set conditions for which fields the validator class applies.
# This is convinience and in fact does the same as an @adapter decorator
# on the PhoneNumberValidator class with the needed interfaces/classes
validator.WidgetValidatorDiscriminators(
    PhoneNumberValidator,
    field=IZohoContactForm['phone_number']
)
```

In `configure.zcml` add an adapter registration like so:

```
<adapter factory=".myform.PhoneNumberValidator" />
```

More info

- original documentation: [z3c.form validators documentation](#).
- <http://docs.plone.org/develop/addons/schema-driven-forms/customising-form-behaviour/validation.html#field-widget-validators>
- <http://www.jowettenterprises.com/blog/an-image-dimension-validator-for-plone-4>

Custom field specific validation in form action handlers and update()

- <http://stackoverflow.com/a/17466776/315168>

Customizing and translating error messages

If you want to custom error messages on per-field level:

```
from zope.schema._bootstrapinterfaces import RequiredMissing
RequiredMissingErrorMessage = error.ErrorViewMessage(_(u'Required value is missing.'),
    ↪ error=RequiredMissing, field=IEmailFormSchema['email'])
zope.component.provideAdapter(RequiredMissingErrorMessage, name='message')
```

Leave `field` parameter out if you want the new error message to apply to all fields.

Read-only and disabled fields

Read-only fields are not rendered in form edit mode:

```
courseModeAccordion = schema.TextLine(
    title=u"Courses by mode accordion",
    default=u"Automatically from database",
    readonly=True
)
```

If the widget mode is `display` then it is rendered as in form view mode, so that the user cannot edit:

```
form.mode(courseModeAccordion="display")
courseModeAccordion = schema.TextLine(
    title=u"Courses by mode accordion",
    default=u"Automatically from database",
)
)
```

Files and images

Description

How to program files and image fields for `z3c.forms` and Dexterity content types

Introduction

This chapter discuss about file uploads and downloads using `zope.schema` based forms and content with *Dexterity content subsystem*.

Note: These instructions apply for Plone 4 and forward. These instructions does not apply for Archetypes content or PloneFormGen.

Plone uses “blobs” (large binary objects) to store file-like data in the ZODB. The ZODB writes these objects to the filesystem as separate files, but due to security, performance and transaction consideration, the original filename is not visible. The files are stored in a distributed tree.

For more introduction information, see:

- Dexterity developer manual

Simple content item file or image field

- – Dexterity developer manual

Simple upload form example

We use `plone.namedfile` for the upload field, which is a CSV file. We accept the upload and then process the file.

You need to declare an `extends` directive to pin down required dependency versions in `buildout.cfg`. For more information, see [buildout troubleshooting](#).

You also need to declare the following packages as dependencies in the `install_dependencies` directive of your `setup.py` file:

- `plone.autoform`,
- `plone.directives.form`.

After doing this, rerunning `buildout` will pull in these packages for you and you will be able to import them successfully. For more information, see [plone.directives.form README](#).

Open the `configure.zcml` file and add register the view:

```
<configure
  xmlns="http://namespaces.zope.org/zope"
  xmlns:browser="http://namespaces.zope.org/browser"
  xmlns:plone="http://namespaces.plone.org/plone"
  i18n_domain="example.dexterityforms">

  ...

  <browser:page
    for="Products.CMFCore.interfaces.ISiteRoot"
    name="import_companies"
    permission="cmf.ManagePortal"
    class=".importusers.ImportUsersForm"
  />

</configure>
```

Create a module named `importusers.py`, and add the following code to it:

```
# -*- coding: utf-8 -*-

# Core Zope 2 + Zope 3 + Plone
from zope.interface import Interface
from zope import schema
from zope.component.hooks import getSite
from Products.CMFCore.interfaces import ISiteRoot
from Products.CMFCore.utils import getToolByName
from Products.CMFCore import permissions
from Products.CMFPlone import PloneMessageFactory as _
from Products.statusmessages.interfaces import IStatusMessage

# Form and validation
from z3c.form import field
import z3c.form.button
from plone.directives import form
import plone.autoform.form

import StringIO
import csv

from plone.namedfile.field import NamedFile
from plone.i18n.normalizer import idnormalizer
```

```

class IImportUsersFormSchema(form.Schema):
    """ Define fields used on the form """

    csv_file = NamedFile(title=_(u"CSV file"))

class ImportUsersForm(form.SchemaForm):
    """ A sample form showing how to mass import users using an uploaded CSV file.
    """

    # Form label
    name = _(u"Import Companies")

    # Which plone.directives.form.Schema subclass is used to define
    # fields for this form
    schema = IImportUsersFormSchema

    ignoreContext = True

    def processCSV(self, data):
        """
        """
        io = StringIO.StringIO(data)

        reader = csv.reader(io, delimiter=',', dialect="excel", quotechar='"')

        header = reader.next()
        print header

    def get_cell(row, name):
        """ Read one cell on a

        @param row: CSV row as list

        @param name: Column name: 1st row cell content value, header
        """

        assert type(name) == unicode, "Column names must be unicode"

        index = None
        for i in range(0, len(header)):
            if header[i].decode("utf-8") == name:
                index = i

        if index is None:
            raise RuntimeError("CSV data does not have column:" + name)

        return row[index].decode("utf-8")

    # Map CSV import fields to a corresponding content item AT fields
    mappings = {
        u"Puhnro" : "phonenumber",
        u"Fax" : "faxnumber",
        u"Postinnumero" : "postalCode",
        u"Postitoimipaikka" : "postOffice",
        u"Www-osoite" : "homepageLink",
        u"Lähiosoite" : "streetAddress",
    }

```

```
        }

    updated = 0

    for row in reader:

        # do stuff ...
        updated += 1

    return updated

@z3c.form.button.buttonAndHandler(_('Import'), name='import')
def importCompanies(self, action):
    """ Create and handle form button "Create company"
    """

    # Extract form field values and errors from HTTP request
    data, errors = self.extractData()
    if errors:
        self.status = self.formErrorsMessage
        return

    # Do magic
    file = data["csv_file"].data

    number = self.processCSV(file)

    # If everything was ok post success note
    # Note you can also use self.status here unless you do redirects
    if number is not None:
        # mark only as finished if we get the new object
        IStatusMessage(self.request).addStatusMessage(_(u"Created/updated_
↪companies:") + unicode(number), "info")
```

File field contents

Example:

```
from zope import schema
from zope.interface import implements, alsoProvides
from persistent import Persistent
from plone import namedfile
from plone.namedfile.field import NamedBlobFile, NamedBlobImage
from zope.schema.fieldproperty import FieldProperty

class IHeaderAnimation(form.Schema):
    """ Alternative header flash animation/imagae """

    animation = NamedBlobFile(title=u"Header flash animation", description=u"Upload_
↪SWF file which is shown in the header", required=False)

# Sample file data used in simulated uploads
sample_data = (
```

```
'GIF89a\x10\x00\x10\x00\xd5\x00\x00\xff\xff\xff\xff\xff\xfe\xfc\xfd\xfd'
'\xfa\xfb\xfc\xfd\xfe\xff\xfa\xfb\xfc\xfd\xfe\xff\xfa\xfb\xfc\xfd\xfe'
'\xeb\xfb\xfc\xfd\xfe\xff\xeb\xfb\xfc\xfd\xfe\xff\xeb\xfb\xfc\xfd\xfe'
'\xd2\xdf\xeb\xfd\xfe\xff\xcd\xdc\xeb\xfb\xfd\xfe\xff\xcd\xdc\xeb\xfb'
'\xc6\xdc\xdc\xcd\xdc\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd'
'\xbd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd'
'\xb6\xcc\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd'
'\xb0\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd'
'\xa8\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd'
'\x9b\xbf\xcc\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd'
'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
'\x00,\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
'\xd4\x84\x01\x01\xe1\xf0d\x16\x9f\x80A\x01\x91\xc0ZmL\xb0\xcd\x00V\x04'
'\xc4a\x87z\xed\x0-\x1a\x03\x08\x95\x0df8\x1e\x11\xca,MoC$\x15\x18{'
'\x006}m\x13\x16\x1a\x1f\x83\x85}6\x17\x1b $\x83\x00\x86\x19\x1d!%) \x8c'
'\x866#\'+.\x8ca` \x1c` (,/1\x94B5\x19\x1e"&*~024\xacNq\xba\xbb\x08h\x0eb'
'\x00A\x00;'
)
```

```
class HeaderAnimation(Persistent):
    """ Persistent storage object used in IHeaderBehavior.alternatives list.

    This holds information about one animation/image upload.
    """
    implements(IHeaderAnimation)

    animation = FieldProperty(IHeaderAnimation["animation"])

animation = HeaderAnimation()
animation.file = namedfile.NamedBlobFile(sample_data, filename=u"flash.swf")
```

Connstring download URLs

Simple example

In Dexterity you can specify a @@download field for content types:

```
<!-- Render link to video file if it's uploaded to this context item -->
<tal:video define="video nocall:context/videoFile"
    tal:condition="nocall:video">
    <a class="flow-player" tal:attributes="href string:${context/absolute_url}/
    ↪@@download/videoFile/${video/filename}"></a>
</tal:video>
```

Complex example

You need to expose file content to the site user through a view and then refer to the URL of the view in your HTML template. There are some tricks you need to keep in mind:

- All file download URLs should be timestamped, or the reupload file change will not be reflected in the browser.
- You might want to serve different file types from different URLs and set special HTTP headers for them.

Complex example (plone.app.headeranimations):

```
from plone.namedfile.interfaces import INamedBlobFile, INamedBlobImage

# <browser:page> providing blob object traverse and streaming
# using download_blob() function below
download_view_name = "@@header_animation_helper"

def construct_url(context, animation_object_id, blob):
    """ Construct download URL for delivering files.

    Adds file upload timestamp to URL to prevent cache issues.

    @param context: Content object who own the files

    @param animation_object_id: Unique identified for the animation in the animation_
    ↪container
        (in the case there are several of them)

    @param field_value: NamedBlobFile or NamedBlobImage or None

    @return: None if there is no blob or the blob field value is empty (file has been_
    ↪removed from admin interface)
        """

    if blob == None:
        return None

    # This case occurs when the file has been removed thorough form interfaces
    # (one of keep, replace, remove options on file widget)

    if animation_object_id == None:
        raise RuntimeError("Cannot have None id")

    # Timestamping prevents caching issues,
    # otherwise the browser shows the old version after reupload
    if hasattr(blob, "_p_mtime"):
        # Zope persistency timestamp is float seconds since epoch
        timestamp = blob._p_mtime
    else:
        timestamp = ""

    # We have different BrowserView methods for download depending on the file type
    # (to apply Flash fix)
    if INamedBlobFile.providedBy(blob):
        func_name = "download_animation"
    else:
        func_name = "download_image"

    # This looks like
    return context.absolute_url() + "/" + download_view_name + "/" + func_name + "?
    ↪timestamp=" + str(timestamp)
```

Streaming file data

File data is delivered to the browser as a stream. The view function returns a streaming iterator instead of raw data. This greatly reduces the latency and memory usage when the file should not be buffered as a whole to memory before

sending.

Example (plone.app.headeranimation):

```
from zope.publisher.interfaces import IPublishTraverse, NotFound

from plone.namedfile.utils import set_headers, stream_data
from plone.namedfile.interfaces import INamedBlobFile, INamedBlobImage

def download_blob(context, request, file):
    """ Stream animation or image BLOB to the browser.

    @param context: Context object name is used to set the filename if blob itself_
    ↪ doesn't provide one

    @param request: HTTP request

    @param file: Blob object
    """
    if file == None:
        raise NotFound(context, '', request)

    # Try determine blob name and default to "context_id_download"
    # This is only visible if the user tried to save the file to local computer
    filename = getattr(file, 'filename', context.id + "_download")

    # Sets Content-Type and Content-Length
    set_headers(file, request.response)

    # Set headers for Flash 10
    # http://www.littled.net/new/2008/10/17/plone-and-flash-player-10/
    cd = 'inline; filename=%s' % filename
    request.response.setHeader("Content-Disposition", cd)

    return stream_data(file)

class HeaderAnimationFieldDownload(BrowserView):
    """ Allow file and image downloads in form widgets.

    Unlike HeaderAnimationHelper, this does not do
    any kind of header resolving, but serves files always
    from the context object itself.
    """

    def __init__(self, context, request):
        self.context = context
        self.request = request
        self.behavior = IHeaderBehavior(self.context)

        self.animation_object_id = self.request.form["animation_object_id"]

    def lookUpAnimation(self):
        """ Don't do look-up in init, since failure there will raise_
        ↪ ComponentLookupError instead of NotFound.

        @return: Blob object to be streamed
        """
        if not self.animation_object_id in self.behavior.alternatives:
```

```

        raise NotFound(self, "Bad animation id:" + self.animation_object_id ,
↳self.request)

    return self.behavior.alternatives[self.animation_object_id]

def download_animation(self):
    """ """
    animation = self.lookupAnimation()
    return download_blob(self.context, self.request, animation.animation)

def download_image(self):
    """ """
    animation = self.lookupAnimation()
    stream_iterator = download_blob(self.context, self.request, animation.image)
    return stream_iterator

```

POSKeyError on missing blob

A `POSKeyError` is raised when you try to access blob *attributes*, but the actual file is not available on the disk. You can still load the blob object itself fine (as it's being stored in the ZODB, not on the filesystem).

Example:

```

Module ZPublisher.Publish, line 119, in publish
Module ZPublisher.mapply, line 88, in mapply
Module ZPublisher.Publish, line 42, in call_object
Module plone.app.headeranimation.browser.views, line 92, in download_image
Module plone.app.headeranimation.browser.views, line 75, in _download_blob
Module plone.app.headeranimation.browser.download, line 90, in download_blob
Module plone.namedfile.utils, line 58, in stream_data
Module ZODB.Connection, line 811, in setstate
Module ZODB.Connection, line 876, in _setstate
Module ZODB.blob, line 623, in loadBlob
POSKeyError: 'No blob file'

```

This might occur for example because you have copied the `Data.fs` file to another computer, but not blob files.

You probably want to catch `POSKeyError`s and return something more sane instead:

```

def download_blob(context, request, file):
    """ Stream animation or image BLOB to the browser.

    @param context: Context object name is used to set the filename if blob itself_
↳doesn't provide one

    @param request: HTTP request

    @param file: Blob object
    """

    from ZODB.POSException import POSKeyError
    try:
        if file == None:
            raise NotFound(context, '', request)

        # Try determine blob name and default to "context_id_download"
        # This is only visible if the user tried to save the file to local computer

```



```

filename = getattr(file, 'filename', context.id + "_download")

set_headers(file, request.response)

# Set headers for Flash 10
# http://www.littled.net/new/2008/10/17/plone-and-flash-player-10/
cd = 'inline; filename=%s' % filename
request.response.setHeader("Content-Disposition", cd)

return stream_data(file)
except POSKeyError:
    # Blob storage damaged
    logger.warn("Could not load blob for " + str(context))
    raise NotFound(context, '', request)

```

See also

- <https://pypi.python.org/pypi/experimental.gracefulblobmissing/>

Widget download URLs

Some things you might want to keep in mind when playing with forms and images:

- Image data might be incomplete (no width/height) during the first POST.
- Image URLs might change in the middle of request (image was updated).

If your form content is something else than traversable context object then you must fix file download URLs manually.

Migrating custom content for blobs

Some hints how to migrate your custom content:

- <http://plone.293351.n2.nabble.com/plone-4-upgrade-blob-and-large-files-tp5500503p5500503.html>

Form encoding

Warning: Make sure that all forms containing file content are posted as `enctype="multipart/form-data"`. If you don't do this, Zope decodes request POST values as string input and you get either empty strings or filenames as your file content data. The older `plone.app.z3cform` templates do not necessarily declare `enctype`, meaning that you need to use a custom page template file for forms doing uploads.

Example correct form header:

```

<form action="." enctype="multipart/form-data" method="post" tal:attributes="action_
↪ request/getURL">

```

File-system access in load-balanced configurations

The `plone.namedfile` product page contains configuration instructions for `plone.namedfile` and ZEO.

WYSIWYG text editing and TinyMCE

Description

WYSIWYG text field editor programming in Plone.

Introduction

Plone supports TinyMCE (default), and CKEditor and others through external add-ons.

In Plone 5, TinyMCE and the Plone integration is provided by the [Mockup project](#).

Disabling HTML filtering and safe HTML transformation

By default Plone does HTML filtering to prevent [cross-site scripting](#) attacks. This will make Plone to strip away from HTML

- `<script>` tags
- Some other potentially unsafe tags and attributes

If you need to put a `<script>` tag on your content text in TinyMCE you can disable this security feature.

Warning: If you don't trust all of your site editors, then this will open your site for an attack.

Step 1: Turn off Plone's `safe_html` transform. Go to `/portal_transforms/safe_html` in the Management Interface, and enter a 1 in the 'disable_transform' box. This prevents Plone from removing tags and attributes while rendering rich text.

Step 2: Set the "X-XSS-Protection: 0" response header. This can be done in your frontend webserver such as apache or nginx. Alternatively, if you only need to disable the protection for users who have permission to edit, you can add this to the site's `main_template`:

```
tal:define="dummy python:checkPermission('Modify portal content', context) and_
↳ request.RESPONSE.setHeader('X-XSS-Protection', '0');"
```

Step 3: Add script tag to the list of `extended_valid_elements` of TinyMCE. Go to the Control Panel, TinyMCE settings, Advanced tab. Add to the Other settings field:

```
{"extended_valid_elements": "script[language|type|src]"}
```

More info

- https://www.tinymce.com/docs/configure/content-filtering/#extended_valid_elements
- <http://glicksoftware.com/blog/disable-html-filtering>

Content linking

Plone offers many kind of support and enhancements in site internal content linking

- Delete protection: *warning if you try to delete content which is being referred.*

- Migrating of links when the content is being moved

The recommended method for linking the content is *Linking by UID* since *Products.TinyMCE* version 1.3.

- When the text is saved in TinyMCE all relative links are converted to *UID links* in the saved HTML payload
- When the text is displayed again, the HTML is run through output filter and UID links are converted back to human readable links

This solves issues with earlier Plone versions where the link targets become invalid when a HTML textfield with relative links were shown on the other page as the original context.

Note: You might need to turn on *Linking by UID* setting on in the site setup if you are migrating from older Plone sites.

Editor preferences

Plone supports user text changeable editor. The active editor is stored in the *user preferences*.

The user can fallback to hand-edited HTML by setting active editor to none.

The rich text widget can also support optional input formats besides HTML: structured text and so on.

Text format selector

The format selector itself is rendered by `wysiwyg_support.pt` macros which is Plone core

- https://github.com/plone/Products.CMFPlone/blob/master/Products/CMFPlone/skins/plone_wysiwyg/wysiwyg_support.pt

Applying styles only edit view

You can use TinyMCE body selector make your CSS class have different styles in view and edit modes (inside TinyMCE)

```
/* Break columns in two column layout
 *
 * https://developer.mozilla.org/en/css3_columns
 *
 */

.column-breaker {
    column-break-before: always;
    display: block;
}

.mce-content-body .column-breaker {
    color: red;
    border: 1px dashed red;
    display: block;
}
```

Note: Firefox does not actually support column breaks, so this was useful headaching experience.

Customizing TinyMCE options

Plone 4 uses TinyMCE 3. Plone 5 upgraded to TinyMCE 4, which works with a new concept called formats and therefore a new syntax for inline styles: *Your Custom Format's Title|custom_format_id|custom_icon_id*.

Note: The icon id will be suffixed and used as a CSS class, so you can hook styles to the *.mce-ico.mce-i-custom_icon_id* selector. For block styles there are no icon hooks so you register them similarly to inline styles but omitting the last part, that is, the icon). That's different from Plone 4's *tinymce.xml*, where you specify *Your Custom Format's Title|tag|custom-css-class*.

This means that after defining styles by associating format titles and ids, you need to define each format by using the *Formats* field. There's already a default JSON structure, so if you add your custom entry after *discreet*, you will end up with:

```
{
  "clearfix": {
    "classes": "clearfix",
    "block": "div"
  },
  "discreet": {
    "inline": "span",
    "classes": "discreet"
  },
  "custom_format_id": {
    "block": "div",
    "classes": "custom-css-class additional-class-1 additional-class-2"
  }
}
```

Available format options are described in <https://www.tinymce.com/docs/configure/content-formatting/#formatparameters>

In your add-on code, all TinyMCE options in the control panel can be exported and imported *using GenericSetup, portal_setup and registry.xml*. For instance, you could add the following records to your *registry.xml*:

```
<records interface="Products.CMFPlone.interfaces.ITinyMCESchema" prefix="plone">
  <value key="block_styles" purge="False">
    <element>Your Custom Format's Title|custom_format_id</element>
  </value>
  <value key="inline_styles" purge="False">
    <element>Your Custom Format's Title|custom_format_id|custom_format_id</element>
  </value>
  <value key="formats">
    {
      "clearfix": {
        "block": "div",
        "classes": "clearfix"
      },
      "discreet": {
        "inline": "span",
        "classes": "discreet"
      },
      "custom_format_id": {
        "block": "div",
        "classes": "custom-css-class"
      }
    }
  </value>
</records>
```

```

    }
  </value>
</records>

```

Alternatively you can define “Quick access custom formats”, namely those accessible directly in the first level of the *Formats* menu (instead of inside of *Inline* or *Blocks* styles submenus). You can do this by providing information in the more generic *Other Settings* field, located in the TinyMCE’s control panel *Advanced* tab, instead of in the *formats* field, so ending up with:

```

<records interface="Products.CMFPlone.interfaces.ITinyMCESchema" prefix="plone">
  <value key="other_settings">
    {
      "style_formats": [
        {
          "title": "Quick access custom format",
          "inline": "span",
          "attributes": {
            "class": "custom-css-class"
          }
        }
      ],
      "style_formats_merge": "True"
    }
  </value>
</records>

```

Rich text transformations

- </external/plone.app.dexterity/docs/advanced/rich-text-markup-transformations>
- <https://pypi.python.org/pypi/plone.app.textfield>

Hacking TinyMCE JavaScript

All JavaScript is built and compiled with Plone 5’s new Resource Registry.

TinyMCE plug-ins

The TinyMCE control panel has the ability to provide custom plugins. Custom plugins map to the http://www.tinymce.com/wiki.php/Configuration:external_plugins setting.

A value is in the format of “plugin name|path/to/javascript.js”.

TinyMCE 3 plugins should still work as Plone ships with the TinyMCE backward compatibility layer for TinyMCE 3.

Adding a new plug-in

Here are instructions how to add new plugins to TinyMCE

Plug-in configuration goes to `registry.xml` GS profile with the record:

```
<record name="plone.custom_plugins"
  interface="Products.CMFPlone.interfaces.controlpanel.ITinyMCESchema"
  field="custom_plugins">
  <field type="plone.registry.field.List">
    <value_type type="plone.registry.field.TextLine" />
  </field>
  <value>
    <element>myplugin|some/path/to/script.js</element>
  </value>
</record>
```

Customizing existing plugin

- Go to the Resource Registry control panel
- Click the Overrides tab
- Use the search to find the plugin code you want to override
- Save your changes
- Click the Registry tab
- Click the build button next to the plone-logged-in bundle

Overriding plug-in resources

You can also override CSS, HTML (.htm.pt templates) with `z3c.jbot` as instructed above.

Example:

```
jbot/Products.CMFPlone.static.components.tinymce-built.js.tinymce.plugins.autosave.
↪ plugin.js
```

Warning: Since there resources are loaded in built into one JavaScript file, any change this way will require you to re-build the JavaScript.

Creating forms through-the-web without programming

PloneFormGen

Description

PloneFormGen allows you to build and maintain convenience forms through Plone edit interface.

Introduction

PloneFormGen is a Plone add-on that provides a generic Plone form generator using fields, widgets and validators from Archetypes. Use it to build simple, one-of-a-kind, web forms that save or mail form input.

To build a web form, create a form folder, then add form fields as contents. Individual fields can display and validate themselves for testing purposes. The form folder creates a form from all the contained field content objects.

Final disposition of form input is handled via plug-in action products. Action adapters included with this release include a mailer, a save-data adapter that saves input in tab-separated format for later download, and a custom-script adapter that makes it possible to script simple actions without recourse to the Management Interface.

To make it easy to get started, newly created form folders are pre-populated to act as a simple e-mail response form.

- [PloneFormGen product page](#)
- [PloneFormGen documentation and tutorials](#)
- [Creating forms with PloneFormGen add-on without programming](#)

ZODB, persistency and transactions

ZODB Database

Description

Plone uses the ZODB object database to store its data. The ZODB can act independently in-process, clustered over network or over another database engine, like SQL.

Introduction

Plone uses the ZODB database. The ZODB happily stores any Python object with any attributes — there is no need to write database schema or table descriptions as there is with SQL-based systems. If data models are described somehow the descriptions are written in Python, usually using `zope.schema` package.

This chapter is about the basics of the ZODB, working with the ZODB database directly, like tuning database settings.

More information about ZODB

- <http://www.zodb.org/>
- [Documentation](#)
- [API documentation](#)

Database files

Usually Plone's database is configured to file `var/filestorage/Data.fs` and uploaded files can be found as BLOBs in `var/blobstorage`.

Object database features

The ZODB is an object database. It makes very easy to store different kinds of contentish data in a graph, supporting subclassing (something which SQL often does poorly).

Since the database stores objects, and the objects are defined in Python code, you always need the corresponding Python source code to instantiate the objects stored inside the ZODB. This might feel awkward at first, but you need to have MySQL running to read what's inside MySQL files stored on your disk and so on ...

Warning: The ZODB database is not usable without the Python source code used to create the data. The data is not readable using any SQL-based tools, and there exist little tools to deal with the raw data. The way to access Plone data is running Plone itself and performing queries through it.

Warning: Since correct source code is needed to read ZODB data, this poses a problem for versioning. Even if you use the correct add-on product with proper source code, if the source code version is wrong, it might not work. Data model attributes might be added, modified or deleted between source code revisions, making data operations on the existing database fail by raising Python exceptions (`AttributeError`, `KeyError`).

To work around the ZODB interoperability problems, products like *ore.contentmirror* exist to duplicate Plone content data to read-only SQL database.

Query and searching

ZODB does not provide query services as is i.e. there is no `SELECT` statement.

Plone provides *cataloging* service for this purpose.

This gives some benefits

- You define yourself how data is indexed
- The backend to perform queries is flexible - you can plug-in custom indexes
- `portal_catalog` default catalog is used for all content items to provide basic CMS functionality
- You can have optimized catalogs for specialized data (e.g. reference look-ups using `reference_catalog`)

Data model

There is no hardwired way for describe data in ZODB database.

Subclasses of ZODB `persistent.Persistent` class will have all their attributes and referred objects written to the database using Python pickle mechanism. Lists and dictionaries will be automatically converted to persistent versions.

There are currently three primary ways to define data models in Plone

- Using `zope.schema` package (modern way) to describe Python object properties
- Using Archetypes content type subsystem (all Plone 3 content)
- Not defining the model, but relying on ad hoc object attributes

Read about *zope.schema* how to define a model for the data to be stored in ZODB database.

Transactions and committing

This [in-depth SO answer](#) explains how committing works in ZODB.

- Savepoints and optimism regarding them
- `PersistentList` and list differences when saving data

Browsing

You can explore ZODB with-in Plone using [ZODBBrowser](#).

Packing database

As ZODB is append-only database it remembers all its history unless packed. Packing will erase undo history.

- Why you need to regularly pack ZODB database to keep the performance up
- Packing is similar to VACUUM in PostgreSQL

Packing through-the-web

Manual packing can be executed through Zope Control Panel (not Plone control panel) in Zope application server root (not Plone site root) in the Management Interface.

Packing from command line

`plone.recipe.zeoserver` buildout recipe provides command called `bin/zeopack` inside buildout. It allows you to trigger packing from the command line when Zope is clustered ZEO configuration. `zeopack` command runs against an on-line site.

This command is useful to run in cron to keep your `Data.fs` file growing forever. You can control the number of days of history to be kept, etc., using buildout recipe variables.

More info

- <https://github.com/plone/plone.recipe.zeoserver>

Packing the database offline

See this [blog post](#).

Example how to pack a copy of `Data.fs` in offline using Python snippet:

```
import time
import ZODB.FileStorage
import ZODB.serialize

storage=ZODB.FileStorage.FileStorage('/tmp/Data.fs.copy')
storage.pack(time.time(), ZODB.serialize.referencesf)
```

As this depends on ZODB egg, the easiest way to run the snippet is to `zopepy` script from your buildout/bin folder:

```
bin/zopepy pack.py
```

For more information, see *command-line scripts*.

Visualizing object graphs

- <http://glicksoftware.com/blog/visualizing-the-zodb-with-graphviz>

Cache size

- [Understanding ZODB cache size option](#)

Integrity checks

Especially when you back-up a Data.fs file, it is useful to run integrity checks for the transferred files.

ZODB provides scripts `fstest` and `fsrefs` to check if Data.fs data is intact and there are no problems due to low level disk corruption or bit flip.

- <http://wiki.zope.org/ZODB/FileStorageBackup>

Note: It is recommended best practice to run integrity against your Data.fs regularly. This is the only way to detect corruption which would otherwise go unnoticed for a long time.

Restart and cache warm-up

Discussion why Plone is slow after restart

- <https://mail.zope.org/pipermail/zodb-dev/2013-March/014935.html>

Recovering old data

Instructions for undoing deleted data and fixing broken databases.

- <http://www.zopatista.com/plone/2008/12/18/saving-the-day-recovering-lost-objects>

ZODB tips and tricks

Please see

- <https://plone.org/events/regional/nola05/collateral/Chris%20McDonough-ZODB%20Tips%20and%20Tricks.pdf>

Persistent objects

Description

This document tells how to save objects to Plone/Zope database. Persistent objects are automatically read and written from ZODB database in Plone and they appear as normal Python objects in your code. This document clarifies some of special properties, like with containers, when you deal with persistent objects programmatically.

Introduction

Q: How do I save() object in Plone

A: You don't

Plone does this automatically for you. You just assign the file data as an attribute of some persistent object. When the HTTP request completes, Zope transaction manager will automatically update all changed persistent objects to the database. There is no “save” as such in Zope world - it all is transparent to the developer. If the transaction fails in any point, no data is being written and you do not need to worry about the partial data being written to the database.

- Changed objects will be automatically saved (if they are attached to the traversing graph)
- Save will not occur if an exception is raised

If your data class inherits from higher level Plone base classes (all go up to `persistent.Persistent` class). persistency is handled transparently for you. Plone also handles transaction automatically for each HTTP request. Unless you wish to do manual transactions there is no need to call `transaction.commit()`.

If you want to do your own persistent classes please read the following

- [Writing a persistent class](#)
- [About persistent objects](#)
- [Persistent interface description.](#)
- [ZODB tips and tricks](#)

Lists and dictionaries

If you modify objects inside persistent lists and dictionaries, the change is not automatically reflected to the parent container.

- [Modifying mutable objects](#)

PersistentList vs. normal Python list

All items in normal Python list are stored as one write and loaded on one write. `PersistentList` is slower, but allows individual objects picked from the list without loading the whole list.

For more information, see

- <https://mail.zope.org/pipermail/zodb-dev/2009-December/013011.html>

Persistent, modifications, `__setattr__` and transactions

When Persistent object is modified, via attribute set or `__setattr__()` call, the current transaction is converted to a write transaction. Write transactions are usually undoable (visible on Zope's Undo tab).

If you are using Python property mutator and even if it does not write to the object it still will trigger the object rewrite.

More info

- <https://mail.zope.org/pipermail/zodb-dev/2009-December/013047.html>

Up-to-date reads

Normally, ZODB only assures that objects read are consistent, but not necessarily up to date. Checking whether an object is up to date is important when information read from one object is used to update another.

The following will force the object to use the most up-to-date version in the transaction:

```
self._p_jar.readCurrent(ob)
```

A conflict error will be raised if the version of ob read by the transaction isn't current when the transaction is committed.

Note: ZODB versions older than 3.10.0b5 do not support this feature.

More information

- <https://pypi.python.org/pypi/ZODB3/3.10.0b5#b5-2010-09-02>

Accessing broken objects

ZODB is object database. By default, it cannot load object from the database if the code (Python class) is not present.

You can still access data in the objects by creating Python code “stubs” which fake the non-existing classes in the run-time environment.

More info

- <http://mockit.blogspot.com/2010/11/getting-broken-objects-out-of-zodb.html>

Fixing damaged objects

If your BTrees have been damaged, you can use `dm.historical` tool to inspect the object history and rewind it to a working state.

- <http://plone.293351.n2.nabble.com/Cleaning-up-damaged-BTree-can-t-delete-folder-tp5761780p5773269.html>
- <https://pypi.python.org/pypi/dm.historical/>

See also

- *Deleting broken objects*

Volatile references

Volatile attributes are attributes on persistent objects which never get stored. ZODB assumes variable is volatile if it has `_v_` prefix.

Volatiles are useful when framework expects the object to conform certain interface, like form frameworks. However, your persistent object edited by form cannot have persistent attributes for all variables the form expects to see.

Example:

```

from persistent import Persistent
from zope.annotation import IAnnotations

class VolatileContext(object):
    """ Mix-in class to provide context variable to persistent classes which is not
    ↪persistent.

    Some subsystems (e.g. forms) expect objects to have a reference to parent/site/
    ↪whatever.
    However, it might not be a wise idea to have circular persistent references.

    This helper class creates a context property which is volatile (never persistent),
    but can be still set on the object after creation or after database load.
    """

    def _set_context(self, context):
        self._v_context = context

    def _get_context(self):
        return self._v_context

class MobileBehaviorStorage(VolatileContext, Persistent):
    """Set mobile specific field properties on the context object and return the
    ↪context object itself.#

    This allows to use attribute storage with schema input validation.
    """

    mobileFolderListing = FieldPropertyDelegate(IMobileBehavior["mobileFolderListing
    ↪"])

KEY = "mobile"

def manufacture_mobile_behavior(context):

    annotations = IAnnotations(context)
    if not KEY in annotations:
        annotations[KEY] = MobileBehaviorStorage()

    object = annotations[KEY]

    # Set volatile context
    object.context = context

    return object

```

Correct use of volatile variables in functions

WRONG:

```

if hasattr(self, '_v_image'):
    return self._v_image

```

RIGHT:

```
marker = []
value = getattr(self, "_v_image", marker)
if value is not marker:
    return value
```

RIGHT:

```
try:
    return self._v_image
except AttributeError:
```

WRONG:

```
self._v_image=expensive_calculation()
return self._v_image
```

RIGHT:

```
image=expensive_calculation()
self._v_image=image
return image
```

For more information, see

- <https://mail.zope.org/pipermail/zodb-dev/2010-May/013437.html>

Measuring persistent object sizes

Get the size of the pickled object in the database.

Something like:

```
pickle, serial = obj._p_jar._storage.load(obj._p_oid, obj._p_jar._version)
```

See also

- <http://blog.hannosch.eu/2009/05/visualizing-persistent-structure-of.html>
- <https://plone.org/documentation/kb/debug-zodb-bloat>
- `treeanalyze.py` will give you the total size of a traverse graph <http://svn.erp5.org/erp5/trunk/utils/treenalyser.py?view=markup&pathrev=24405>

Transactions

Introduction

Plone uses the **ZODB** database which implements **Multiversion concurrency control**.

Plone will complete either *all* database modifications that occur during a request, or *none* of them. It will never write incomplete data to the database.

Plone and the underlying Zope handles transactions transparently.

Note: Every transaction is a *read* transaction until any of the objects participating in the transaction are mutated (object attribute set), turning the transaction to a *write* transaction.

Note: Old examples might refer to the `get_transaction()` function. This has been replaced by `transaction.get()` in the later Zope versions.

Please read this [Zope transaction tutorial](#) to get started how to use transactions with your code.

- <https://bugs.launchpad.net/zope2/+bug/143584>

Using transactions

Normally transactions are managed by Plone and the developer should not be interested in them.

Special cases where one would want to manage transaction life-cycle may include:

- Batch creation or editing of many items once.

Example code:

- [transaction source code](#).
- <http://www.zope.org/Members/mcdonc/HowTos/transaction>
- <https://bugs.launchpad.net/zope3/+bug/98382>

Subtransactions

Normally, a Zope transaction keeps a list of objects modified within the transaction in a structure in RAM.

This list of objects can grow quite large when there is a lot of work done across a lot of objects in the context of a transaction. *Subtransactions* write portions of this object list out to disk, freeing the RAM required by the transaction list. Using subtransactions can allow you to build transactions involving objects whose combined size is larger than available RAM.

Example:

```
import transaction
...

done = 0
for brain in all_images:
    done += 1
    ...
    # Since this is HUGE operation (think resizing 2 GB images)
    # it is not nice idea to buffer the transaction (all changed data)
    # in the memory (Zope default transaction behavior).
    # Using subtransactions we hint Zope when it would be a good time to
    # flush the changes to the disk.
    if done % 10 == 0:
        # Commit subtransaction for every 10th processed item
        transaction.get().commit(True)
```

Failsafe crawling and committing in batches

In the case you need to access many objects in coherent and efficient manner.

- <https://bitbucket.org/gocept/gocept.linkchecker/src/80a127405ac06d2054e61dd62fcd643d864357a0/src/gocept/linkchecker/scripts/crawl-site.py?at=default>

Transaction boundary events

It is possible to perform actions before and after transaction is written to the database.

See transaction documentation about [before commit hooks](#) and [after commit hooks](#).

Viewing transaction content and debugging transactions

Please see *Transaction troubleshooting*

Undoing transactions

Everything that has happened on Plone site can be undone through the *Undo* tab in the Management Interface, in the site root. By default you can undo latest 20 transactions.

If you need to raise this limit just replace all numbers of 20 with higher value in file `App/Undo.py`, restart site and now you can undo more transactions.

Object lifecycles

Plone has different lifecycles for different objects

- Persistent objects: These objects are transparently persistent. They look like normal Python objects, but they are serialized to the disk if the transaction completes successfully. Persistent object inherit from Zope's various persistent classes: `persistent.Persistent`, `PersistentDict`, `PersistentList` and they have special attributes like `_p_mtime` when the object was last written to disk. To make object persistent, it must be referred from Zope's App traversing graph. Examples: content objects, user account objects.
- Request attached objects and thread-local objects: Each HTTP request is processed by its own Python thread. These objects disappear when the request has been processed. Examples: request object itself, `getSite()` thread-local way to access the site object, request specific permission caches.
- In-process objects, or "static" objects are created when the server application is launched and they are gone when the application quits. Usually these objects are set-up during Plone initialization and they are read-only for served HTTP requests. Examples: content type vocabulary lists.

Storage

Description

What kind of different storages (storing backends) ZODB has and how to use them.

Introduction

This page explains the details how data is stored in the ZODB. The information here is important to understand Plone's database behavior and how to optimize your application.

Pickling

ZODB is an object oriented database. All data in the ZODB are stored as [pickled Python objects](#). Pickle is the object serialization module in Python's standard library.

- Each time an object is read and it is not cached, object is read from the ZODB data storage and unpickled
- Each time an object is written, it is pickled and transaction machinery appends it to the ZODB data storage

Pickle format is a series of bytes. Here is an example what it looks like:

```
>>> import pickle
>>> data = { "key" : "value" }
>>> pickled = pickle.dumps(data)
>>> print pickled
(dp0
S'key'
p1
S'value'
p2
s.
```

It is not a very human readable format.

Even if you use SQL based [RelStorage](#) ZODB backends, the objects are still pickled to the database; SQL does not support varying table schema per row and Python objects do not have fixed schema format.

Binary trees

Data is usually organized to binary trees or [BTrees](#). More specifically, data is usually stored as Object Oriented Binary Tree [OOBtree](#) which provides a Python object as key and Python object value mappings. Key is the object id in the parent container as a string, and value is any pickleable Python object or primitive you store in your database.

[ZODB data structure interfaces](#).

[Using BTrees example from Zope Docs](#).

Buckets

BTree stores data in buckets ([OOBucket](#)).

A Bucket is the smallest unit of data which is written to the database once. Buckets are loaded lazily: BTree only loads buckets storing values of keys being accessed.

BTree tries to put as much data as possible into one bucket. When one value in a bucket is changed, the whole bucket must be rewritten to the disk.

[Default bucket size is 30 objects](#).

Storing as attribute vs. storing in BTree

Plone has two kinds of fundamental way to store data:

- Attribute storage (stores values directly in the pickled objects).
- Annotation storage (OOBTree based). Plone objects have attribute `__annotations__` which is an OOBtree for storing objects in a name-conflict free way.

When objects are stored in the annotation storage, reading object values needs at least one extra database look up to load the first bucket of the OOBTree.

If the value is going to be used frequently, and especially if it is read when viewing the content object, storing it in an attribute is more efficient than storing it in an annotation. This is because the `__annotations__` BTree is a separate persistent object which has to be loaded into memory, and may push something else out of the ZODB cache.

If the attribute stores a large value, it will increase memory usage, as it will be loaded into memory each time the object is fetched from the ZODB.

BLOBs

BLOBs are large binary objects like files or images.

BLOBs are supported since ZODB 3.8.x.

When you use BLOB interface to store and retrieve data, they are stored physically as files on your file systems. A file system, as the name says, was designed to handle files and has far better performance on large binary data than sticking the data into ZODB.

BLOBs are streamable which means that you can start serving the file from the beginning of the file to HTTP wire without needing to buffer the whole data to the memory first (slow).

SQL values

Plone's Archetypes subsystem supports storing individual Archetypes fields in SQL database. This is mainly an [integration feature](#). Read more about this in [Archetypes manual](#).

Transaction sizes

Discussion pointers

- <http://www.mail-archive.com/zodb-dev@zope.org/msg03398.html>

Analysing Data.fs content offline

- <https://plone.org/documentation/kb/debug-zodb-bloat>

Migrations

Database migrations are needed if your internal data storage format changes between versions.

ZODB does not require you to set object format explicitly, like in SQL you need to create table schema. However, your code will naturally fail if the data format of the object is unexpected.

- [Changing instance attributes](#)

Functionality and features

Description

Explanations how specific user visible features are programmed in Plone.

Actions

Description

Creating and using portal_actions mechanism

Introduction

Plone has concept of actions which connect the end user functionality associated with site or content objects:

- View, edit, sharing etc. are actions
- Sitemap is action
- Contact form is action
- Cut, copy, paste are actions
- Logged in menu is populated by actions

Actions are managed by

- portal_actions for generic actions
- portal_types for view, edit etc. actions and object default action... all actions which are tied to a particular content type and may vary by type

Iterating through available actions

Here is a page template example

```
<ul>
  <tal:actions repeat="action python:context.portal_actions.
  ↳listFilteredActionsFor(context) ['portal_tabs']">
    <li>
      <a tal:attributes="href action/url; title action/title;" tal:content=
      ↳"action/title">
        Action title
      </a>
    </li>
  </tal:actions>
</ul>
```

Creating actions through-the-web

You can manage the actions from Site Setup using the Actions control panel.

This control panel lists all the existing actions, grouped by category. Any action can be modified (using the **Edit** button), or removed (using the **Delete** button). The **Add** button allows to create a new action.

To re-order actions, click **Edit** and change the position parameter value.

To move an action to another category, click **Edit** and change the category parameter value.

Exporting and importing all portal_actions

You can transfer action configuration from a Plone site to another using GenericSetup export/import XML. You can also do this to generate XML from which you can cut out snippets for creating actions.xml by hand.

- Go to portal_setup
- Choose Export
- Choose actions
- Choose “Export selected steps” button at the end of the page
- ...and so on

Creating actions.xml by hand

Usually all actions are rewritten by site policy product using portal_actions import/export. Actions are in GenericSetup profile file *default/profiles/actions.xml*.

1. actions.xml is exported from the development instance using portal_setup
2. actions.xml is made part of the site policy product

Alternatively, if you are developing add-on product, you can add actions one-by-one by manually creating entries in actions.xml.

Example how to add an action to the document_actions (like rss and print):

```
<?xml version="1.0"?>
<object name="portal_actions" meta_type="Plone Actions Tool"
  xmlns:i18n="http://xml.zope.org/namespaces/i18n">
  <object name="document_actions" meta_type="CMF Action Category">
    <object name="sendto" meta_type="CMF Action" i18n:domain="plone">
      <property name="title" i18n:translate="">Send this</property>
      <property name="description" i18n:translate=""></property>
      <property name="url_expr">string:$object_url/sendto_form</property>
      <property name="icon_expr"></property>
      <property name="available_expr">object/@@shareable</property>
      <property name="permissions">
        <element value="Allow sendto"/>
      </property>
      <property name="visible">True</property>
    </object>
  </object>
</object>
```

Example how to add actions to user menu, which is visible in the top right corner for logged in users (Plone 4):

```
<?xml version="1.0"?>
<object name="portal_actions" meta_type="Plone Actions Tool"
  xmlns:i18n="http://xml.zope.org/namespaces/i18n">
  <object name="user" meta_type="CMF Action Category">
    <object name="ora_sync" meta_type="CMF Action" i18n:domain="plone">
      <property name="title" i18n:translate="">ORA</property>
      <property name="description" i18n:translate="">ORA site synchronization status</
    <property>
```

```

<property name="url_expr">string:${portal_url}/@@syncall</property>
<property name="icon_expr"></property>
<property name="available_expr"></property>
<property name="permissions">
  <element value="Manage portal"/>
</property>
<property name="visible">True</property>
</object>
</object>
</object>

```

Reordering actions in actions.xml

Try using these attributes

- insert-after
- insert-before

They accept * and action name parameters.

Example:

```

<object name="sendto" meta_type="CMF Action" i18n:domain="plone" insert-before="*">

```

Action URLs

Actions are applied to objects by adding action name to url.

E.g.:

```
http://localhost:8080/site/page/view
```

for view action and:

```
http://localhost:8080/site/page/edit
```

for edit action.

Action can be also not related to document, like:

```
http://localhost:8080/site/sitemap
```

Default action

Default action is executed when the content URL is opened without any prefix.

Default action is defined in portal_types.

Default action can be dynamic - meaning that site editor may set it from Display menu. For more information see Dynamic Views.

Content type specific actions

Content type specific actions can be registered in `portal_types`. Actions are viewable and editable in the Management Interface under `portal_types`. After editing actions, content type XML can be exported and placed to your content type add-on product.

GenericSetup example file for content type “ProductCard” which has a new tab added next to view, edit, sharing, etc. File is located in `profiles/default/types/ProductCard.xml`.

```
<?xml version="1.0"?>
<object name="ProductCard"
  meta_type="Factory-based Type Information with dynamic views"
  i18n:domain="saariselka.app" xmlns:i18n="http://xml.zope.org/namespaces/i18n">
  <property name="title" i18n:translate="">Tuotekortti</property>
  ....
  <alias from="(Default)" to="(dynamic view)" />
  <alias from="edit" to="atct_edit" />
  <alias from="sharing" to="@@sharing" />
  <alias from="view" to="(selected layout)" />
  <action title="View" action_id="view" category="object" condition_expr=""
    url_expr="string:${object_url}/" visible="True">
    <permission value="View" />
  </action>
  <action title="Edit" action_id="edit" category="object" condition_expr=""
    url_expr="string:${object_url}/edit" visible="True">
    <permission value="Modify portal content" />
  </action>

  <!-- Custom action code goes here. We add a new tab with title "Data" and
    uri @@productdata_view which is a registered BrowserView for the content_
  -->
  <action title="Data" action_id="productdata_view" category="object" condition_expr=""
    url_expr="string:${object_url}/@@productdata_view" visible="True">
    <permission value="Modify portal content" />
  </action>
</object>
```

The corresponding BrowserView is registered as any other view in `browser/configure.zcml`:

```
<browser:page
  for="*"
  name="productdata_view"
  class=".productdataview.ProductDataView"
  template="productdataview.pt"
  allowed_attributes="renderData"
  permission="zope2.View"
/>
```

Toggling action visibility programmatically

Warning: This applies only for Plone 2.5. You should use actions.xml instead.

Example:

```
def disable_actions(portal):
    """ Remove unneeded Plone actions

    @param portal Plone instance
    """

    # getActionObject takes parameter category/action id
    # For ids and categories please refer to portal_actions in the Management Interface
    actionInformation = portal.portal_actions.getActionObject("document_actions/rss")

    # See ActionInformation.py / ActionInformation for available edits
    actionInformation.edit(visible=False)
```

Visibility expressions

In portal_actions expression is used to determine whether an action is visible on a particular page.

Expression is “expression” field in actions.xml or “Expression” field in portal_actions.

Note: This check is just a visibility check. Users can still try to type the action by typing the URL manually. You need to do the permission level security check on the view providing the action.

For more information see [expressions](#).

Condition examples

See in [expressions](#).

Using actions in views and viewlets

Example:

```
context_state = getMultiAdapter((self.context, self.request),
                                name=u'plone_context_state')

# First argument is action category,
# we have custom "mobile_actions"
self.actions = context_state.actions().get('mobile_actions', None)
```

Tabs (sections)

Tabs are special actions

- Some of tabs are automatically generated from root level content items
- Some of tabs are manually added to `portal_actions.portal_tabs`

By default, they are shown as the top vertical navigation of Plone site.

Example how to generate tabs list:

```
def getSections(self):
    """

    @return: tuple (selectedTabs, currentSelectedTab)
    """

    context_state = getMultiAdapter((self.context, self.request),
                                    name=u'plone_context_state')
    actions = context_state.actions()

    # Get CatalogNavigationTabs instance
    portal_tabs_view = getMultiAdapter((self.context, self.request),
                                       name='portal_tabs_view')

    # Action parameter is "portal_tabs" by default, but can be other
    portal_tabs = portal_tabs_view.topLevelTabs(actions=actions)

    selectedTabs = self.context.restrictedTraverse('selectedTabs')

    selected_tabs = selectedTabs('index_html',
                                self.context,
                                portal_tabs)

    selected_portal_tab = selected_tabs['portal']

    return (portal_tabs, selected_portal_tab)
```

Custom action listings

Example:

```
import Acquisition
from zope.component import getMultiAdapter

class Sections(base.Sections):
    """
    """

    def update(self):
        base.Sections.update(self)

        context = Acquisition.aq_inner(self.context)
        # IContextState view provides shortcut to get different action listings
        context_state = getMultiAdapter((context, self.request), name=u'plone_context_
↪state')
        all_actions = context_state.keyed_actions() # id -> action mappings
        mobile_site_actions = all_actions["mobile_site_actions"].values()
        self.portal_tabs = mobile_site_actions
```


Different tabs per section/folder

You might want to have different actions for different site sections or folders.

- <http://plone.293351.n2.nabble.com/Custom-portal-tabs-per-subsection-tp5747768p5747768.html>

Copy, cut and paste

These action are based on OFS Zope 2 package SimpleItem mechanisms. Plone specific event handlers are used to update Plone related stuff like `portal_catalog` on move.

Plone internal clipboard relies on the presence of Zope 2 session (different from authentication session). Paste action fails silently (is missing) if `_ZopeId` session cookie does not work correctly on your web server.

Expressions

Description

Expressions are string templates or Python expressions which are used in various places in Plone for templates, action conditions and URL generation.

Introduction

Expressions are part of *TAL*, the Template Attribute Language. They are used in Zope Page Templates (*ZPT*) and as part of workflow definitions, among other things.

You might want to use expressions in your own add-on product to provide user-written conditions for viewlet visibility, portlets, dynamic text, etc.

The authoritative reference is [Appendix C: Zope Page Templates Reference](#) of the *Zope 2 Book*

Expressions are used in:

- the `tal:condition`, `tal:content`, `tal:replace`, `tal:attribute`, `tal:define` *TAL* directives;
- `portal_css`, `portal_javascript` and other resource managers, to express when a resource should be included or not;
- `portal_actions` to define when content, site and user actions are visible.

Expression types

There are three main categories of expressions.

Expression can contain an optional `protocol:` prefix to determine the expression type.

path expression (default)

Unless you specify an expression type using `python:` or `string:` notation, a *path expression* is assumed.

Path expressions use slashes for traversal (*traversing*), and will implicitly call callables.

Example: call the `Title()` method on the `context` object (finding it by *acquisition* if necessary) and return its value:

```
context/Title
```

Variables can be included using `?`. Example: access a folder using the id stored in the `myItemId` variable, and return its title:

```
context/?myItemId/Title
```

Note: With this kind of usage, if the variable you’re dereferencing isn’t sanitized, there could be security ramifications. Use `python:restrictedTraverse()` instead if you need to use variables in your path parts.

`__call__()` and `nocall`: behavior in TAL path traversing

The TAL path expression will call Python callable objects by default.

If you try to get a hold of a *helper view* like this:

```
tal:define="commentsView context/@@comments_view"
```

You might get this exception:

```
Module zope.tales.expressions, line 217, in __call__
Module Products.PageTemplates.Expressions, line 155, in _eval
Module Products.PageTemplates.Expressions, line 117, in render
Module Products.Five.browser.metaconfigure, line 476, in __call__
AttributeError: 'coments_view' object has no attribute 'index'
```

It basically means that your view does not have a template assigned and the traversing logic tries to render that template.

This happens because

1. “`context/@@comments_view`” creates a view instance
2. then calls its `__call__()` method
3. the default `BrowserView.__call__()` behavior to render a template by doing:

```
def __call__(self):
    return self.index()
```

4. Because your view does not have a template assigned it also lacks `self.index` attribute

The workaround for cases like this is to use `nocall::` traversing:

```
tal:define="commentsView nocall:context/@@comments_view"
```

string: expressions

Do string replace operation.

Example:

```
string:${context/portal_url}/@@my_view_name
```

python: expression

Evaluate as Python code.

Example:

```
python:object.myFunction() == False
```

Expression variables

Available expression variables are defined in `CMFCore/Expressions.py`:

```
data = {
    'object_url':    object_url,
    'folder_url':    folder.absolute_url(),
    'portal_url':    portal.absolute_url(),
    'object':        object,
    'folder':        folder,
    'portal':        portal,
    'nothing':       None,
    'request':       getattr(portal, 'REQUEST', None),
    'modules':       SecureModuleImporter,
    'member':        member,
    'here':          object,
}
```

You can also access *helper views* directly by name.

Using expressions in your own code

Expressions are persistent objects. You usually want to attach them to something, but this is not necessary.

Example:

```
from Products.CMFCore.Expression import Expression, getExprContext

# Create a sample expression - usually this is taken from
# the user input
expression = Expression("python:context.Title() == 'foo'")

expression_context = getExprContext(self.context)

# Evaluate expression by calling
# Expression.__call__(). This
# will return whatever value expression evaluation gives
value = expression(expression_context)

if value.strip() == "":
    # Usually empty expression field means that
    # expression should be True
    value = True
```

```
if value:
    # Expression succeeded
    pass
else:
    pass
```

Custom expression using a helper view

If you need to add complex Python code to your expression conditions it is best to put this code in a `BrowserView` and expose it as a method.

Then you can call the method on a view from a TALEs expression:

```
object/@@my_view_name/my_method
```

Your view code would look like:

```
class MyViewName(BrowserView):
    """ Exposes methods for expression conditions """

    def my_method(self):
        """ Funky condition

        self.context = object for which this view was traversed
        """
        if self.context.Title().startswith("a"):
            return True
        else:
            return False
```

Register the view as “my_view_name”, using `configure.zcml` as usual.

You can use context interfaces like

- `Products.CMFCore.interfaces.IContentish`
- `zope.interface.Interface` (or `*`)

to make sure that this view is available on all content objects, as TALEs will be evaluated on every page, depending on what kind of content the page will present.

Expression examples

Get current language

Use *IPortalState context helper* view.

Example how to generate a multilingual-aware RSS feed link:

```
string:${object/@@plone_portal_state/portal_url}/site-feed/RSS?set_language=${object/
↪@@plone_portal_state/language}
```

... or you can use a Python expression for comparison:

```
python:object.restrictedTraverse('@@plone_portal_state').language() == 'fi'
```

Check current language in TAL page template

For example, in case you need to generate HTML such as links conditionally, depending on the current language:

Example:

```
<a tal:define="language context/@@plone_portal_state/language" tal:condition="python:
↪language == 'fi'"
    href="http://www.fi">Finnish link</a>
```

Example to have different footers (or something similar) for different languages:

```
<div tal:replace="structure context/footer_text"
    tal:condition="python:context.restrictedTraverse('@@plone_portal_state').
↪language() == 'no'" />
<div tal:replace="structure context/footer_texteng"
    tal:condition="python:context.restrictedTraverse('@@plone_portal_state').
↪language() == 'en'" />
```

Check if object implements an interface

Example:

```
python:context.restrictedTraverse('@@plone_interface_info').provides('Products.
↪CMFCore.interfaces.IFolderish')
```

Returns True or False. Useful for actions.

Check if a certain hostname was used for HTTP request

Example:

```
python:"localhost" in request.environ.get("HTTP_HOST", "")
```

Check if the object is a certain content type

Example:

```
python:getattr(object, "portal_type", "") == "Custom GeoLocation"
```

Get portal description

Example:

```
tal:define="
    portal context/portal_url/getPortalObject;
    portal_description portal/Description"
```

Doing <input CHECKED> and boolean like HTML attributes in TAL

To have a value appear in TAL or not you can do:

```
<input type="checkbox" tal:attributes="checked python:'checked' if MYCONDITION else ''
↪" />
```

We execute a Python snippet which

- We will dynamically create a *checked* attribute on *<input>* based on Python evaluation
- Return “checked” string if some condition we check in Python evaluates to True
- Otherwise we return an empty string and TAL won’t output this attribute (TODO: has TAL some special support for CHECKED and SELECTED attributes)

Note: Python 2.6, Plone 4+ syntax

Through-the-web scripts

The Management Interface allows one to create, edit and execute *RestrictedPython sandboxed scripts* directly through the web management interface. This functionality is generally discouraged nowadays in the favor of *view classes*.

Creating a TTW Python script in an add-on installer

Here is an example of how one can pre-seed a Python script in an add-on installer *GenericSetup profile*.

setuphandlers.py:

```
from Products.PythonScripts.PythonScript import manage_addPythonScript

DEFAULT_REDIRECT_PY_CONTENT = """
if port not in (80, 443):
    # Don't kick in HTTP/HTTPS redirects if the site
    # is directly being accessed from a Zope front-end port
    return None
"""

def runCustomInstallerCode(site):
    """ Run custom add-on product installation code to modify Plone site object and
    ↪others

    Python scripts can be created by Products.PythonScripts.PythonScript.manage_
    ↪addPythonScript

    http://svn.zope.org/Products.PythonScripts/trunk/src/Products/PythonScripts/
    ↪PythonScript.py?rev=114513&view=auto

    @param site: Plone site
    """

    # Create the script in the site root
    id = "redirect_handler"

    # Don't override the existing installation
```

```
if not id in site.objectIds():
    manage_addPythonScript(site, id)
    script = site[id]

    # Define the script parameters
    parameters = "url, port"

    script.ZPythonScript_edit(parameters, DEFAULT_REDIRECT_PY_CONTENT)

def setupVarious(context):
    """
    @param context: Products.GenericSetup.context.DirectoryImportContext instance
    """

    # We check from our GenericSetup context whether we are running
    # add-on installation for your product or any other proudct
    if context.readDataFile('collective.scriptedredirect.marker.txt') is None:
        # Not our add-on
        return

    portal = context.getSite()

    runCustomInstallerCode(portal)
```

See the full example.

Dynamically hiding content menu items

- <http://blog.affinitic.be/2013/03/04/filter-menu-using-a-grok-view/>

Portlets

Description

Programmatic manipulation of portlets in Plone.

Introduction

Portlets are editable boxes in the left and right side bar of Plone user interface. Add-ons allow portlets in other parts in of the user interface too, like above and below the content.

This document is a short introduction. Please visit the [Portlets reference manual](#) for in-depth information.

Related add-ons and packages

You might want to check these before starting to write your own portlet - for ready solution, for examples, for inspiration.

- [Create your own portlet managers with collective.panels](#)
- <https://github.com/collective/collective.portletalias>

- <https://plone.org/products/contentwellportlets>
- <https://github.com/miohtama/imageportlet>
- <https://github.com/collective/collective.cover>

Note: Using `paster` is deprecated instead you should use *[bobtemplates.plone](#)*

Creating a portlet

- You need a paster-compatible product skeleton created using *`paster create -t plone`* or *`paster create -t archetypes`* commands.
- Use project specific paster command *`paster addcontent portlet`* to create a code skeleton for your new portlet.

Deprecated since version may_2015: Use *[bobtemplates.plone](#)* instead

Subclassing a portlet

You can subclass a portlet to create a new portlet type with your enhanced functionality.

- [subclassing portlets](#)

Using z3c.form in portlets

z3c.form is a modern form library for Plone. The out of the box Plone portlets use older *zope.formlib*.

Discussion related to the matter

- <http://stackoverflow.com/questions/5174905/can-i-use-z3c-form-on-plone-portlets-instead-of-zope-formlib>

Overriding portlet rendering

Use `<plone:portletRenderer>` directive. Specify 1) layer, 2) template and/or 3) class 4) portlet interface.

You need `<include package="">` directive for the package whose portlet you are going to override.

```
<configure
  xmlns:plone="http://namespaces.plone.org/plone"
  >

    <include package="plone.app.portlets" />

    <plone:portletRenderer
      portlet="plone.app.portlets.portlets.news.INewsPortlet"
      template="mytheme_news.pt"
      layer=".interfaces.IThemeSpecific"
    />

</configure>
```

More information

update() and render()

These methods should honour `zope.contentprovider.interfaces.IContentProvider` call contract.

available property

The portlet renderer can define available property to hint the portlet manager when the portlet should be rendered.

Example

```
class Renderer(base.Renderer):

    @property
    def available(self):
        # Show this portlet for logged in users only
        return not self.anonymous
```

Iterate portlets assigned to the portal root

Below is a simple example how to print all portlets which have been assigned to the portal root:

```
def check_root_portlets(self):
    """ Print all portlet assignments in the portal root """

    from zope.component import getUtility, getMultiAdapter
    from plone.portlets.interfaces import IPortletManager
    from plone.portlets.interfaces import IPortletAssignment
    from plone.portlets.interfaces import IPortletAssignmentMapping

    content = self.portal

    for manager_name in [ "plone.leftcolumn", "plone.rightcolumn" ]:

        print "Checking portlet column:" + manager_name

        manager = getUtility(IPortletManager, name=manager_name, context=content)

        mapping = getMultiAdapter((content, manager), IPortletAssignmentMapping)

        # id is portlet assignment id
        # and automatically generated
        for id, assignment in mapping.items():
            print "Found portlet assignment:" + id + " " + str(assignment)
```

Looking up a portlet by id

Here are some tips how to extract the portlet id data in the portlet renderer to pass around to be consumed elsewhere.

portlets.py:

```
class Renderer(base.Renderer):

    def getImageURL(self, imageDesc):
        """
```

```
        :return: The URL where the image can be downloaded from.

        """
        context = self.context.aq_inner

        # [{'category': 'context', 'assignment': <imageportlet.portlets.Assignment_
↪ object at 0x1138bb140>, 'name': u'bound-method-assignment-title-of-assignment-at-1',
↪ 'key': '/Plone/fi'}],
        params = dict(
            portletName=self.__portlet_metadata__["name"],
            portletManager=self.__portlet_metadata__["manager"],
            image=imageDesc["id"],
            modified=self.data._p_mtime,
            portletKey=self.__portlet_metadata__["key"],
        )

        imageURL = "%s/@@image-portlet-downloader?%s" % (context.absolute_url(),
↪ urllib.urlencode(params))

        return imageURL
```

Then we can re-look-up this portlet and its image field, based on the field name, in the downloader view:

```
# Zope imports
from zExceptions import InternalError
from zope.interface import Interface
from zope.component import getUtility, getMultiAdapter
from five import grok

# Plone imports
from plone.portlets.interfaces import IPortletManager
from plone.portlets.interfaces import IPortletRetriever
from plone.namedfile.utils import set_headers, stream_data

# Local imports
from interfaces import IAddonSpecific

grok.templatedir("templates")
grok.layer(IAddonSpecific)

class ImagePortletHelper(grok.CodeView):
    """
    Expose stuff downloadable from the image portlet BLOBs.
    """
    grok.context(Interface)
    grok.baseclass()

class ImagePortletImageDownload(ImagePortletHelper):
    """
    Expose image fields as downloadable BLOBS from the image portlet.

    Allow set caching rules (content caching for this view)
    """
    grok.context(Interface)
    grok.name("image-portlet-downloader")
```

```

def getPortletById(self, content, portletManager, key, name):
    """
    :param content: Context item where the look-up is performed

    :param portletManager: Portlet manager name as a string

    :param key: Assignment key... context path as string for content portlets

    :param name: Portlet name as a string

    :return: Portlet assignment instance
    """

    # Make sure we got input
    assert key, "Give a proper portlet assignment key"
    assert name, "Give a proper portlet assignment name"

    # Resolve portlet and its image field
    manager = getUtility(IPortletManager, name=portletManager, context=content)

    # Mappings can be directly used only when
    # portlet is directly assignment to the content.
    # If it is assigned to the parent we would fail here.
    # mapping = getMultiAdapter((content, manager), IPortletAssignmentMapping)

    retriever = getMultiAdapter((content, manager,), IPortletRetriever)

    for assignment in retriever.getPortlets():
        if assignment["key"] == key and assignment["name"] == name:
            return assignment["assignment"]

    return None

def render(self):
    """

    """
    content = self.context.aq_inner

    # Read portlet assignment pointers from the GET query
    name = self.request.form.get("portletName")
    manager = self.request.form.get("portletManager")
    imageId = self.request.form.get("image")
    key = self.request.form.get("portletKey")

    portlet = self.getPortletById(content, manager, key, name)
    if not portlet:
        raise InternalError("Portlet not found: %s %s" % (key, name))

    image = getattr(portlet, imageId, None)
    if not image:
        # Ohops?
        raise InternalError("Image was empty: %s" % imageId)

```

See *imageportlet* add-on for the complete example.

Walking through every portlet on the site

The following code iterates through all portlets assigned directly to content items. This excludes dashboard, group and content type based portlets. Then it prints some info about them and renders them.

Example code:

```
from Products.Five.browser import BrowserView

from zope.component import getUtility, getMultiAdapter
from zope.app.component.hooks import setHooks, setSite, getSite

from plone.portlets.interfaces import IPortletType
from plone.portlets.interfaces import IPortletManager
from plone.portlets.interfaces import IPortletAssignment
from plone.portlets.interfaces import IPortletDataProvider
from plone.portlets.interfaces import IPortletRenderer
from plone.portlets.interfaces import IPortletAssignmentMapping
from plone.portlets.interfaces import ILocalPortletAssignnable

from Products.CMFCore.interfaces import IContentish

class FixPortlets(BrowserView):
    """ Magical portlet debugging view """

    def __call__(self):
        """
        """

        request = self.request

        portal = getSite()

        # Not sure why this is needed...
        view = portal.restrictedTraverse('@@plone')

        # Query all content items on the site which can get portlets assigned
        # Note that this should excule special, hidden, items like tools which
        ↪ otherwise
        # might appearn in portal_catalog queries
        all_content = portal.portal_catalog(show_inactive=True, language="ALL",
        ↪ object_provides=ILocalPortletAssignnable.__identifier__)

        # Load the real object instead of index stub
        all_content = [ content.getObject() for content in all_content ]

        # portal itself does not show up in the query above,
        # though it might contain portlet assignments
        all_content = list(all_content) + [portal]

        for content in all_content:

            for manager_name in [ "plone.leftcolumn", "plone.rightcolumn" ]:

                manager = getUtility(IPortletManager, name=manager_name,
                ↪ context=content)

                mapping = getMultiAdapter((content, manager),
                ↪ IPortletAssignmentMapping)
```

```

        # id is portlet assignment id
        # and automatically generated
        for id, assignment in mapping.items():
            print "Found portlet assignment:" + id + " " +
↳str(assignment)

            renderer = getMultiAdapter((content, request,
↳view, manager, assignment), IPortletRenderer)

            # Renderer acquisition chain must be set-up so
↳that templates

            # et. al. can resolve permission inheritance
            renderer = renderer.__of__(content)

            # See https://github.com/zopefoundation/zope.
↳contentprovider/blob/3.7.2/src/zope/contentprovider/interfaces.py
            renderer.update()
            html = renderer.render()
            print "Got HTML output:" + html

        return "OK"

```

For more information about portlet assignments and managers, see

- https://github.com/plone/plone.app.portlets/blob/master/plone/app/portlets/tests/test_mapping.py
- https://github.com/plone/plone.app.portlets/blob/master/plone/app/portlets/tests/test_traversal.py
- <https://github.com/plone/plone.app.portlets/blob/master/plone/app/portlets/configure.zcml>
- <https://github.com/plone/plone.portlets/blob/master/plone/portlets/interfaces.py>
- <https://github.com/zopefoundation/zope.contentprovider/blob/3.7.2/src/zope/contentprovider/interfaces.py> (for portlet renderers)

Checking if a certain context portlet is active on a page

- Iterate through portlet managers by name
- Get portlet retriever for the manager
- Get portlets
- Check if the portlet assignment provides your particular portlet marker interface

Example:

```

import Acquisition
from zope.component import getUtility, getMultiAdapter

from plone.portlets.interfaces import IPortletRetriever, IPortletManager

for column in ["plone.leftcolumn", "plone.rightcolumn"]:

    manager = getUtility(IPortletManager, name=column)

    retriever = getMultiAdapter((self.context, manager), IPortletRetriever)

```

```
portlets = retriever.getPortlets()

for portlet in portlets:

    # portlet is {'category': 'context', 'assignment': <FacebookLikeBoxAssignment
    ↪at facebook-like-box>, 'name': u'facebook-like-box', 'key': '/isleofback/sisalto/
    ↪huvit-ja-harrasteet
    # Identify portlet by interface provided by assignment
    if IFacebookLikeBoxData.providedBy(portlet["assignment"]):
        return True

return False
```

Rendering a portlet

Below is an example how to render a portlet in Plone

- A portlet is assigned to some context in some portlet manager
- We can dig these assignments up by portlet id (not user visible) or portlet type (portlet assignment interface)

How to get your portlet HTML:

```
from zope.component import getUtility, getMultiAdapter, queryMultiAdapter
from plone.portlets.interfaces import IPortletRetriever, IPortletManager,
    ↪IPortletRenderer
from plone.portlets.interfaces import IPortletManagerRenderer

from Products.Five import BrowserView

class FakeView(BrowserView):
    """
    Portlet manager code goes down well with cyanide.
    """

def get_portlet_manager(column):
    """ Return one of default Plone portlet managers.

    @param column: "plone.leftcolumn" or "plone.rightcolumn"

    @return: plone.portlets.interfaces.IPortletManagerRenderer instance
    """
    manager = getUtility(IPortletManager, name=column)
    return manager

def render_portlet(context, request, view, manager, assignmentId):
    """ Render a portlet defined in external location.

    .. note ::

        Portlets can be idenfied by id (not user visible)
        or interface (portlet class). This method supports look up
```

```

        by interface and will return the first matching portlet with this interface.

@param context: Content item reference where portlet appear

@param manager: IPortletManager instance through get_portlet_manager()

@param view: Current view or None if not available

@param interface: Marker interface class we use to identify the portlet. E.g.
↳ IFacebookPortlet

@return: Rendered portlet HTML as a string, or empty string if portlet not found
"""

if not view:
    # manager(context, request, view) does not accept None as multi-adapter,
↳ lookup parameter
    view = FakeView(context, request)

retriever = getMultiAdapter((context, manager), IPortletRetriever)

portlets = retriever.getPortlets()

assignment = None

if len(portlets) == 0:
    raise RuntimeError("No portlets available for manager %s in the context %s" %
↳ (manager.__name__, context))

for portlet in portlets:

    # portlet is {'category': 'context', 'assignment': <FacebookLikeBoxAssignment
↳ at facebook-like-box>, 'name': u'facebook-like-box', 'key': '/isleofback/sisalto/
↳ huvit-ja-harrasteet
    # Identify portlet by interface provided by assignment
    print portlet
    if portlet["name"] == assignmentId:
        assignment = portlet["assignment"]
        break

if assignment is None:
    # Did not find a portlet
    raise RuntimeError("No portlet found with name: %s" % assignmentId)

# Note: Below is tested only with column portlets

# PortletManager provides convenience callable
# which gives you the renderer. The view is mandatory.
managerRenderer = manager(context, request, view)

# PortletManagerRenderer convenience function
renderer = managerRenderer._dataToPortlet(portlet["assignment"].data)

if renderer is None:
    raise RuntimeError("Failed to get portlet renderer for %s in the context %s"
↳ % (assignment, context))

renderer.update()

```

```
# Does not check visibility here... force render always
html = renderer.render()

return html
```

How to use this code in your own view, please see `collective.portletalias` source

More info

- <http://blog.mfabrik.com/2011/03/10/how%C2%A0to-render-a-portlet-in-plone/>

Hiding unwanted portlets

Example portlets.xml:

```
<!-- This leaves only News portlet -->

<portlet addview="portlets.Calendar" remove="true" />
<portlet addview="portlets.Classic" remove="true" />
<portlet addview="portlets.Login" remove="true" />
<portlet addview="portlets.Events" remove="true" />
<portlet addview="portlets.Recent" remove="true" />
<portlet addview="portlets.rss" remove="true" />
<portlet addview="portlets.Search" remove="true" />
<portlet addview="portlets.Language" remove="true" />
<portlet addview="plone.portlet.collection.Collection" remove="true" />
<portlet addview="plone.portlet.static.Static" remove="true" />

<!-- collective.flowplayer add-on -->
<portlet addview="collective.flowplayer.Player" remove="true" />
```

Portlet names can be found in `plone.app.portlets/configure.zcml`.

More info:

- <http://stackoverflow.com/questions/5897656/disabling-portlet-types-site-wide-in-plone>

Disabling right or left columns in a view or template

Sometimes, when you work with custom views and custom templates you need to disable right or left column for portlets.

This is how you do from within a template:

```
<metal:override fill-slot="top_slot"
  tal:define="disable_column_one python:request.set('disable_plone.leftcolumn',1);
             disable_column_two python:request.set('disable_plone.rightcolumn',1);
  <\/>
```

And this is how you do it from within a view:

```
import grok

class SomeView(grok.View):
    grok.context(IPloneSiteRoot)

    def update(self):
```



```

super(SomeView, self).update()
self.request.set('disable_plone.rightcolumn', 1)
self.request.set('disable_plone.leftcolumn', 1)

```

Source: <http://stackoverflow.com/questions/5872306/how-can-i-remove-portlets-in-edit-mode-with-plone-4>

Disabling right or left columns on a context

Sometimes you just want to turn off the portlets in a certain context that doesn't have a template or fancy view. To do this in code do this:

```

from zope.component import getMultiAdapter
from zope.component import getUtility

from plone.portlets.interfaces import IPortletManager
from plone.portlets.interfaces import ILocalPortletAssignmentManager
from plone.portlets.constants import CONTEXT_CATEGORY

# Get the proper portlet manager
manager = getUtility(IPortletManager, name=u"plone.leftcolumn")

# Get the current blacklist for the location
blacklist = getMultiAdapter((context, manager), ILocalPortletAssignmentManager)

# Turn off the manager
blacklist.setBlacklistStatus(CONTEXT_CATEGORY, True)

```

Or just do it using GenericSetup like a sane person:

- <https://plone.org/documentation/manual/developer-manual/generic-setup/reference/portlets>
- <https://plone.org/products/plone/roadmap/203>

Creating a new portlet manager

If you need additional portlet slots at the site. In this example we use `Products.ContentWellCode` to provide us some facilities as a dependency.

- Create a viewlet which will handle portlet rendering in a normal page mode. Have several portlet slots, a.k.a. wells, where you can drop in portlets. Wells are rendered horizontally side-by-side and portlets going in from top to bottom.
- Register this viewlet in a viewlet manager where you wish to show your portlets on the main template
- Have a management view which allows you to shuffle portlets around. This is borrowed from `Products.ContentWellPortlets`.
- Register portlet wells in `portlets.xml` - note that one management view can handle several slots as in the example below

The code skeleton works against [this Plone add-on template](#).

Example portlet manager viewlets.py:

```

"""

For more information see

```

```
* http://docs.plone.org/5/en/develop/plone/views/viewlets.html

"""

import logging
from fractions import Fraction

# Zope imports
from zope.interface import Interface
from zope.component import getMultiAdapter, getUtility, queryUtility
from five import grok

# Plone imports
from plone.portlets.interfaces import IPortletManager
from plone.app.layout.viewlets.interfaces import IPortalFooter
from Products.CMFCore.utils import getToolByName

# Local imports
from interfaces import IAddonSpecific, IThemeSpecific

grok.templatedir("templates")
grok.layer(IThemeSpecific)

# By default, set context to zope.interface.Interface
# which matches all the content items.
# You can register viewlets to be content item type specific
# by overriding grok.context() on class body level
grok.context(Interface)

logger = logging.getLogger("PortletManager")

class CustomPortletViewlet(grok.Viewlet):
    """ grok viewlet base class for a custom portlet renderer based on Products.
    ↪ContentWellPortlets

    Original code from Products.ContentWellPortlets
    """
    grok.baseclass()

    # Id which we use to store portlets
    name = ""

    # Name of browser view which will render the management interface for portlets
    # in this manager
    manage_view = ""

    # We have 5 portlet slots in this viewlet
    portlet_count = 5

    def update(self):
        context_state = getMultiAdapter((self.context, self.request), name=u'plone_
        ↪context_state')
        self.manageUrl = '%s/%s' % (context_state.view_url(), self.manage_view)

        ## This is the way it's done in plone.app.portlets.manager, so we'll do the_
        ↪same
```

```

        mt = getToolByName(self.context, 'portal_membership')
        self.canManagePortlets = mt.checkPermission('Portlets: Manage portlets', self.
↪context)

    def showPortlets(self):
        return '@@manage-portlets' not in self.request.get('URL')

    def portletManagersToShow(self):
        visibleManagers = []

        for n in range(1, self.portlet_count):
            name = '%s%s' % (self.name, n)

            try:
                mgr = getUtility(IPortletManager, name=name, context=self.context)
            except:
                # In the case we have problems to load portlet manager, do something_
↪about it
                # This is graceful fallback in a situation where 1) add-on is already_
↪installed
                # 2) new portlet code drops in and re-run add-on installer is
                continue

            if mgr(self.context, self.request, self).visible:
                visibleManagers.append(name)

        managers = []
        numManagers = len(visibleManagers)
        for counter, name in enumerate(visibleManagers):
            pos = 'position-%s' % str(Fraction(counter, numManagers)).replace('/', ':')
            width = 'width-%s' % (str(Fraction(1, numManagers)).replace('/', ':') if_
↪numManagers > 1 else 'full')
            managers.append((name, 'cell %s %s %s' % (name.split('.')[0], width,
↪pos)))
        return managers

class ColophonPortlets(CustomPortletViewlet):
    """
    Render a new series of portlets in colophon.
    """

    # This name is used to store portlets,
    # as referred in portlets.xml
    name = 'PortletsColophon'

    # This is custom management URL view for this,
    # registered thru ZCML to point to Products.ContentWellContent manager view class.
    manage_view = '@@manage-portlets-colophon'

    grok.viewletmanager(IPortalFooter)
    grok.template("portlets-colophon")

    # Define a portlet manager declaration
    from Products.ContentWellPortlets.browser.interfaces import IContentWellPortletManager

    class IColophonPortlets(IContentWellPortletManager):
        """

```

```
This viewlet is a place holder to match portlets.xml and portlet management view_
↳together.

    * Manager is referred by name in manage page template

    * portlets.xml refers to this interface

    * provider:ColophonPortlets expression is also used in template to render the_
↳actual porlets
    """
```

Example ZCML bit

```
<!-- Register new portlet management view for our portlet manager -->

<include package ="plone.app.portlets" />

<!--

    The .pt file is customized for the portlet manager name (from portlets.xml)
    and management link.

-->

<browser:page
    name="manage-portlets-colophon"
    for="plone.portlets.interfaces.ILocalPortletAssignable"
    class="plone.app.portlets.browser.manage.ManageContextualPortlets"
    template="templates/manage-portlets-colophon.pt"
    permission="plone.app.portlets.ManagePortlets"
/>
```

The page template for the manager `manage-portlets-colophon.pt` is the following

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:metal="http://xml.zope.org/namespaces/metal"
      xmlns:tal="http://xml.zope.org/namespaces/tal"
      xmlns:i18n="http://xml.zope.org/namespaces/i18n"
      metal:use-macro="context/main_template/macros/master"
>

<head>
  <div metal:fill-slot="javascript_head_slot" tal:omit-tag="">
    <script type="text/javascript"
            tal:attributes="src string:${context/absolute_url}/++resource++manage-
↳ portlets.js">
      </div>
  </head>
  <body class="manage-portlet-well">

    <metal:block fill-slot="top_slot"
                  tal:define="disable_column_one python:request.set('disable_
↳ plone.leftcolumn',1);
                                disable_column_two python:request.set('disable_
↳ plone.rightcolumn',1);" />

    <div metal:fill-slot="main">
```

```

<tal:warning tal:condition="plone_view/isDefaultPageInFolder">
  <dl class="portalMessage warning">
    <dt i18n:translate="message_warning_above_content_area_dt">Is
→this really where you want to add portlets above the content?</dt>
    <dd i18n:translate="message_warning_above_content_area_dd">If you
→add portlets here, they will only appear on this item. If instead you want portlets
→to appear on all items in this folder,
      <a href=""
        tal:attributes="href string:${plone_view/
→getCurrentFolderUrl}/@@manage-portlets-colophon"
        i18n:name="manage-portletsinheader_link">
        <span i18n:translate="add_them_to_the_folder_itself">add
→them to the folder itself</span>
      </a>
    </dd>
  </dl>
</tal:warning>

<h1 class="documentFirstHeading"
  i18n:translate="manage_portlets_in_header">Manage portlets in colophon
</h1>

<p>
  <a href=""
    class="link-parent"
    tal:attributes="href string:${context/absolute_url}"
    i18n:translate="return_to_view">
    Return
  </a>
</p>

<div class="porlet-well_manager">
  <h2 i18n:translate="portlet-well-a">Colophon Portlet Well 1</h2>
  <span tal:replace="structure provider:PortletsColophon1" />
</div>

<div class="porlet-well_manager">
  <h2 i18n:translate="portlet-well-a">Colophon Portlet Well 2</h2>
  <span tal:replace="structure provider:PortletsColophon2" />
</div>

<div class="porlet-well_manager">
  <h2 i18n:translate="portlet-well-a">Colophon Portlet Well 3</h2>
  <span tal:replace="structure provider:PortletsColophon3" />
</div>

<div class="porlet-well_manager">
  <h2 i18n:translate="portlet-well-a">Colophon Portlet Well 4</h2>
  <span tal:replace="structure provider:PortletsColophon4" />
</div>

<div class="porlet-well_manager">
  <h2 i18n:translate="portlet-well-a">Colophon Portlet Well 5</h2>
  <span tal:replace="structure provider:PortletsColophon5" />
</div>

</div>

```

```
</body>
</html>
```

Then we have `portlets-colophon.pt` page template for the viewlet which renders the portlets and related management link

```
<div id="portlets-colophon"
      class="row">

  <tal:block tal:condition="viewlet/showPortlets">
    <tal:portletmanagers tal:repeat="manager viewlet/portletManagersToShow">
      <div tal:attributes="class python:manager[1]"
            tal:define="mgr python:manager[0]"
            tal:content="structure provider:${mgr}" />

    </tal:portletmanagers>

    <div style="clear:both"><!-- --></div>

    <div class="manage-portlets-link"
          tal:condition="viewlet/canManagePortlets">
      <a href=""
          class="managePortletsFallback"
          tal:attributes="href viewlet/manageUrl">
        Add, edit or remove a portlet in <b tal:content="viewlet/name" />
      </a>
    </div>

  </tal:block>

</div>
```

Finally there is `portlets.xml` which lists all the portlet managers and associates them with the used interface

```
<?xml version="1.0"?>
<!-- Set up all the new portlet managers we need above and below the content well -->
<portlets>

  <portletmanager
    name="PortletsColophon1"
    type="youraddon.viewlets.IColphonPortlets"
  />

  <portletmanager
    name="PortletsColophon2"
    type="youraddon.viewlets.IColphonPortlets"
  />

  <portletmanager
    name="PortletsColophon3"
    type="youraddon.viewlets.IColphonPortlets"
  />

  <portletmanager
    name="PortletsColophon4"
    type="youraddon.viewlets.IColphonPortlets"
```

```

/>

<portletmanager
  name="PortletsColophon5"
  type="youraddon.viewlets.IColphonPortlets"
/>

</portlets>

```

More info

- <https://weblion.psu.edu/svn/weblion/weblion/Products.ContentWellPortlets/trunk/Products/ContentWellPortlets/>
- <http://stackoverflow.com/questions/9766744/dynamic-tal-provider-expressions>

Fixing relative links for static text portlets

Note: This should be no longer issue with Plone 4.1 and TinyMCE 1.3+ when using UID links.

Example how to convert links in all static text portlets:

```

from lxml import etree
from StringIO import StringIO
import urlparse
from lxml import html

def fix_links(content, absolute_prefix):
    """
    Rewrite relative links to be absolute links based on certain URL.

    @param html: HTML snippet as a string
    """

    parser = etree.HTMLParser()

    content = content.strip()

    tree = html.fragment_fromstring(content, create_parent=True)

    def join(base, url):
        """
        Join relative URL
        """
        if not (url.startswith("/") or "://" in url):
            return urlparse.urljoin(base, url)
        else:
            # Already absolute
            return url

    for node in tree.xpath('//*[@src]'):
        url = node.get('src')
        url = join(absolute_prefix, url)
        node.set('src', url)
    for node in tree.xpath('//*[@href]'):

```

```
href = node.get('href')
url = join(absolute_prefix, href)
node.set('href', url)

data = etree.tostring(tree, pretty_print=False, encoding="utf-8")

return data
```

Other resources and examples

- [Static text portlet.](#)
- [Templated portlet](#)

Site setup and configuration

Description

How to create settings for your add-on product and how to programmatically add new Plone control panel entries.

Introduction

This documentation tells you how to create new “configlets” to Plone site setup control panel.

Configlets can be created in two ways:

- Using the `plone.app.registry` configuration framework for Plone (recommended);
- Using any *view code*.

`plone.app.registry`

`plone.app.registry` is the state of the art way to add settings for your Plone 4.x+ add-ons.

For tutorial and more information please see the [PyPi](#) page.

Example products:

- <https://pypi.python.org/pypi/collective.gtags>
- <https://plone.org/products/collective.habla>
- <https://pypi.python.org/pypi/collective.xdv>

Minimal example

Below is a minimal example for creating a configlet using `plone.app.registry`.

It is based on the [youraddon](#) template. The add-on package in this case is called `silvuple`.

In `buildout.cfg`, make sure you have the `extends` line for Dexterity (see the [Dexterity installation guide](#)).

`setup.py`:


```
install_requires = [..."plone.app.dexterity", "plone.app.registry"],
```

```
configure.zcml
```

```
<configure
  xmlns="http://namespaces.zope.org/zope"
  xmlns:browser="http://namespaces.zope.org/browser"
  xmlns:plone="http://namespaces.plone.org/plone"
  i18n_domain="example.dexterityforms">

  ...

  <browser:page
    name="silvuple-settings"
    for="Products.CMFPlone.interfaces.IPloneSiteRoot"
    class=".settings.SettingsView"
    permission="cmf.ManagePortal"
  />

</configure>
```

```
settings.py:
```

```
"""
    Define add-on settings.
"""

from zope.interface import Interface
from zope import schema
from Products.CMFCore.interfaces import ISiteRoot
from Products.Five.browser import BrowserView

from plone.z3cform import layout
from plone.directives import form
from plone.app.registry.browser.controlpanel import RegistryEditForm
from plone.app.registry.browser.controlpanel import ControlPanelFormWrapper

class ISettings(form.Schema):
    """ Define settings data structure """

    adminLanguage = schema.TextLine(title=u"Admin language",
                                   description=u"Type two letter language code (admins always use this_
↳ language)")

class SettingsEditForm(RegistryEditForm):
    """
    Define form logic
    """
    schema = ISettings
    label = u"Silvuple settings"

class SettingsView(BrowserView):
    """
    View which wrap the settings form using ControlPanelFormWrapper to a HTML_
↳boilerplate frame.
    """
```

```
def render(self):
    view_factor = layout.wrap_form(SettingsEditForm, ControlPanelFormWrapper)
    view = view_factor(self.context, self.request)
    return view()
```

profiles/default/controlpanel.xml

```
<?xml version="1.0"?>
<object
  name="portal_controlpanel"
  xmlns:i18n="http://xml.zope.org/namespaces/i18n"
  i18n:domain="silvuple">

  <configlet
    title="Silvuple Settings"
    action_id="silvuple.settings"
    appId="silvuple"
    category="Products"
    condition_expr=""
    url_expr="string:${portal_url}/@@silvuple-settings"
    icon_expr=""
    visible="True"
    i18n:attributes="title">
      <permission>Manage portal</permission>
    </configlet>

</object>
```

profiles/default/registry.xml

```
<registry>
  <records interface="silvuple.settings.ISettings" prefix="silvuple">
    <!-- Set default values -->

    <!-- Leave to empty string -->
    <value key="adminLanguage"></value>
  </records>
</registry>
```

Control panel widget settings

plone.app.registry provides the RegistryEditForm class, which is a subclass of z3c.form.form.Form.

It has two places to override which widgets will be used for which field:

- updateFields() may set widget factories, i.e. widget type, to be used;
- updateWidgets() may play with widget properties and widget values shown to the user.

Example (collective.gtags project, controlpanel.py):

```
class TagSettingsEditForm(controlpanel.RegistryEditForm):

    schema = ITagSettings
    label = _(u"Tagging settings")
```

```
description = _(u>Please enter details of available tags")

def updateFields(self):
    super(TagSettingsEditForm, self).updateFields()
    self.fields['tags'].widgetFactory = TextLinesFieldWidget
    self.fields['unique_categories'].widgetFactory = TextLinesFieldWidget
    self.fields['required_categories'].widgetFactory = TextLinesFieldWidget

def updateWidgets(self):
    super(TagSettingsEditForm, self).updateWidgets()
    self.widgets['tags'].rows = 8
    self.widgets['tags'].style = u'width: 30%;'
```

plone.app.registry imports — backwards compatibility

You need this if you started using `plone.app.registry` before April 2010.

There is a change concerning the 1.0b1 codebase:

```
try:
    # plone.app.registry 1.0b1
    from plone.app.registry.browser.form import RegistryEditForm
    from plone.app.registry.browser.form import ControlPanelFormWrapper
except ImportError:
    # plone.app.registry 1.0b2+
    from plone.app.registry.browser.controlpanel import RegistryEditForm
    from plone.app.registry.browser.controlpanel import ControlPanelFormWrapper
```

Configlets without `plone.registry`

Just add `controlpanel.xml` pointing to your custom form.

Content type choice setting

Often you need to have a setting whether a certain functionality is enabled on particular content types.

Here are the ingredients:

- A custom schema-defined interface for settings (`registry.xml` schemas don't support multiple-choice widgets in `plone.app.registry 1.0b2`);
- a vocabulary factory to pull friendly type information out of `portal_types`.

`settings.py`:

```
"""
    Define add-on settings.
"""

from zope import schema
from five import grok
from Products.CMFCore.interfaces import ISiteRoot
from zope.schema.interfaces import IVocabularyFactory
```

```

from z3c.form.browser.checkbox import CheckBoxFieldWidget

from plone.z3cform import layout
from plone.directives import form
from plone.app.registry.browser.controlpanel import RegistryEditForm
from plone.app.registry.browser.controlpanel import ControlPanelFormWrapper

class ISettings(form.Schema):
    """ Define settings data structure """

    adminLanguage = schema.TextLine(title=u"Admin language", description=u"Type two_
↳ letter language code and admins always use this language")

    form.widget(contentTypes=CheckBoxFieldWidget)
    contentTypes = schema.List(title=u"Enabled content types",
                               description=u"Which content types appear on_
↳ translation master page",
                               required=False,
                               value_type=schema.Choice(source="plone.app.
↳ vocabularies.ReallyUserFriendlyTypes"),
                               )

class SettingsEditForm(RegistryEditForm):
    """
    Define form logic
    """
    schema = ISettings
    label = u"Silvuple settings"

class SettingsView(grok.CodeView):
    """

    """
    grok.name("silvuple-settings")
    grok.context(ISiteRoot)
    def render(self):
        view_factor = layout.wrap_form(SettingsEditForm, ControlPanelFormWrapper)
        view = view_factor(self.context, self.request)
        return view()

```

profiles/default/registry.xml:

```

<registry>
  <records interface="silvuple.settings.ISettings" prefix="silvuple.settings.
↳ ISettings">
    <!-- Set default values -->

    <value key="contentTypes" purge="false">
      <element>Document</element>
      <element>News Item</element>
      <element>Folder</element>
    </value>
  </records>

```

```
</registry>
```

Configuring Plone products from buildout

See a section in the [Buildout](#) chapter

Configuration using environment variables

If your add-on requires “setting file” for few simple settings you can change for each buildout you can use operating system environment variables.

For example, see:

- <https://pypi.python.org/pypi/Products.LongRequestLogger>

Dashboard

Introduction

Dashboard is a “block” in user preferences.

Tutorial

- <https://plone.org/documentation/kb/setup-a-custom-dashboard>

RSS

Description

Programming RSS feeds on Plone sites

local

- *RSS*
 - *Introduction*
 - *Creating a global, language neutral, Plone site content aggregator*
 - * *Creating the collection*
 - * *Collecting content for the RSS feed*
 - * *Linking the RSS feed to site action links*
 - * *Publish and test*
 - *Syndication Settings*
 - * *Plone <= 4.2*

- * *Plone >= 4.3*
- *Publishing content through RSS in Plone 4*
- *RSS feed content*
- *Changing RSS feed template*
- * *Enabling full body text in RSS feed*

Introduction

Plone can generate RSS feeds from folderish content types (folder / collection). If you want to aggregate content from all the site to RSS feed, you first create a collection content item and then enable RSS feed on this collection content item.

Creating a global, language neutral, Plone site content aggregator

These instructions tell you how to create a RSS feed collection for your Plone site. You can choose what content types ends up to the RSS stream. Also, the aggregator is language aware so that it works correctly on multilingual sites.

Creating the collection

First we create a collection which will aggregate all the site content for the RSS feed.

- Go to site root
- Add new collection
- Title “Your site name - RSS feed”
- On *Category* tab, set *Language* to neutral
- On *Settings* tab, choose *Exclude from navigation*
- Save
- Go to site root / *Contents* tab
- Check your RSS content collection
- Choose *Rename* button
- Change item id to `site-feed`

Collecting content for the RSS feed

- Go to your collection content item
- Go to *criteria* tab
- Set *content types* criteria
- Set sort by publishing date, reverse
- Save
- Now, choose content items you want to appear in the feed and *Save* again

You can now preview the content of RSS feed on *View* tab.

Linking the RSS feed to site action links

Site actions is the top right link slot on the Plone site. By default, Plone site wide RSS link will appear there if enabled.

- Go to `portal_actions` in the Management Interface
- Go to `/portal_actions/site_actions/rss`
- In URL expression type:

```
string:${object}/@@plone_portal_state/portal_url}/site-feed/RSS?set_language=${
↪{object}/@@plone_portal_state/language}
```

This expression will

- Get URL for *site-feed* object, using *RSS* template
- Will explicitly set HTTP GET query parameter *set_language* which can be used to manually force Plone content language. We use the current language (from the user cookie) here, to make sure that the user gets RSS feed in correct language on multilingual sites.

More about *expressions*.

Publish and test

Publish collection after the content seems to be right, using the workflow menu on the collection content item.

Test RSS feed by copy-pasting RSS URL from the site action to your RSS Reader, like *Google Reader*.

Syndication Settings

Plone <= 4.2

`portal_syndication` is a persistent utility managing RSS settings. It provides settings to for formatting RSS feeds (frequency of updates, number of items).

- <https://github.com/plone/Products.CMFPlone/blob/4.2.x/Products/CMFPlone/SyndicationTool.py>

Plone >= 4.3

In Plone 4.3, the `portal_syndication` utility was replaced by a browser view and registry settings.

The view may be traversed to from any context with `@@syndication-util`.

for example, in Plone 4.2 you check for the ability to syndicate a context like so:

```
<p class="discreet"
  tal:condition="context/portal_syndication/isSiteSyndicationAllowed">
  <a href=""
    class="link-feed"
    i18n:translate="title_rss_feed"
    tal:define="here_url context/@@plone_context_state/object_url"
    tal:attributes="href string:$here_url/search_rss?${request/QUERY_STRING}">
```

```
</p>
    Subscribe to an always-updated feed of these search terms</a>
```

In Plone 4.3, this is updated to look like this:

```
<p class="discreet"
  tal:condition="context/@@syndication-util/search_rss_enabled">
  <a href=""
    class="link-feed"
    i18n:translate="title_rss_feed"
    tal:define="here_url context/@@plone_context_state/object_url"
    tal:attributes="href string:$here_url/search_rss?${request/QUERY_STRING}">
    Subscribe to an always-updated feed of these search terms</a>
</p>
```

The `syndication-util` view is found in `Products.CMFPlone.browser.syndication.utils`

- <https://github.com/plone/Products.CMFPlone/blob/master/Products/CMFPlone/browser/syndication/utils.py>

Publishing content through RSS in Plone 4

Access `/content/synPropertiesForm` and publish.

RSS feed content

RSS feed content is the content of the folder or special stream provided by the content type.

`portal_syndication` uses the following logic to pull the content:

```
if hasattr(obj, 'synContentValues'):
    values = obj.synContentValues()
else:
    values = obj.getFolderContents()
return values
```

Changing RSS feed template

RSS feed is stored in template `CMFPlone/skins/plone_templates/rss_template`.

Enabling full body text in RSS feed

See [this example](#).

Collections

Description

Collections are site editor enabled searches. They provide automatic, folder like view, for the content fetched from the Plone site by criteria defined by the site editor.

Introduction

Note: In Plone 4.2, old style collections have been replaced with new style collections, featuring a vastly improved user interface and a de-coupling from the ATTopic content type (i.e. they no longer use ATTopic).

They are useful to generate different listings.

Collections are internally called “topics” and the corresponding content type is “ATTopic” (< 4.2 only). Collections were renamed from topics in Plone 3.0.

Collection searches are driven by two factors:

- User visible “criteria” which is mapped to portal_catalog queries
- portal_catalog() indexes which you need to add yourself for custom content types. Read more about them in *Searching and Indexing chapter*

Add new collection criteria (new style, plone.app.collection installed)

How to add your own criteria to a collection plone.app.collection and (or more precisely the underlying plone.app.querystring) uses plone.app.registry records to define possible search criteria for a collection.

If you want to add your own criteria, say to choose a value from a custom index, you have to create a plone.app.registry record for this index in your generic setup profile (e.g profiles/default/registry.xml):

```
<registry>
  <records interface="plone.app.querystring.interfaces.IQueryField"
    prefix="plone.app.querystring.field.department">
    <value key="title">Department</value>
    <value key="description">A custom department index</value>
    <value key="enabled">True</value>
    <value key="sortable">False</value>
    <value key="operations">
      <element>plone.app.querystring.operation.string.is</element>
    </value>
    <value key="group">Metadata</value>
  </records>
</registry>
```

The title-value refers to the custom index (“Department”), the operations-value is used to filter the items and the group-value defines under which group the entry shows up in the selection widget.

Note

For a full list of all existing QueryField declarations see <https://github.com/plone/plone.app.querystring/blob/master/plone/app/querystring/profiles/default/registry.xml#L197>

For a full list of all existing operations see <https://github.com/plone/plone.app.querystring/blob/master/plone/app/querystring/profiles/default/registry.xml#L1>

Adding new collection criteria (old style, < 4.2 only)

portal_catalog search indexes are not directly exposed to the collection criteria management backend, since portal_catalog indices do not support features like localization and user-friendly titles.

Note: In Plone 4.2, the Collection configlet is no longer listed in Site Setup. But you can still access it here: http://localhost:8080/Plone/portal_atct/atct_manageTopicIndex.

New criteria can be created through-the-web in Site setup -> Collection section. Click “All fields” to see unenabled portal_catalog criteria. Later the edited settings can be exported to GenericSetup XML profile using portal_setup tool (no need to create profile XML files by hand).

portal_catalog indices can be added through-the-web (TTW) through the Management Interface portal_catalog tool tabs.

If you still want to create XML files by hand, read more about it in [Enable Collection Indices \(fields for searching\)](#) for custom types [HOW TO](#).

Sticky sorting

See:

- <http://stackoverflow.com/questions/8791132/how-to-create-sticky-news-items-in-plone-4>

Locked content

Since Plone 3.1 content edit locking has been supported. This feature is to prevent simultaneous conflicting edits of the same content.

If the editor forgot to press Save or Cancel explicit unlocking must be performed on locked objects if you want to modify them. Unfortunately the side-effect is that if site has objects left to locked state they usually interfere with your programming.

Since Plone 3.3 the feature can be disabled from the site settings.

Unlocking content

Unlocking can be performed from the object view or edit tab.

Unlocking from Zope control panel

- <http://destefano.wordpress.com/2009/01/15/more-fun-with-plone-locks/>

Scripts to unlock all site content

- <http://m10880.kaivo.com/Plone/help-center/how-to/unlock-webdav-locks>
- <http://www.zopelabs.com/cookbook/1002703851>

Breadcrumbs (path bar)

Description

Breadcrumbs is visual element showing where the user is on the site. This document shows some example code how to create breadcrumbs programmatically.

Navigation level sensitive breadcrumbs

Below is a breadcrumbs viewlet displayed only on 3rd navigation level downwards. Drop this in your [add-on template](#). Tune the `visible()` function for further functionality.

Python code to be dropped in `viewlets.py`:

```
from plone.app.layout.viewlets.interfaces import IAboveContent

class Breadcrumbs(grok.Viewlet):
    """ Breadcrumbs override which are only displayed on 2nd level and forward (not_
    ↪ on Home screen)
    """

    # Override standard Plone breadcrumbs
    grok.name("plone.path_bar")
    grok.viewletmanager(IAboveContent)

    def visible(self):
        """ Called by template condition. """

        # Note that "Home" does not count as a crumb
        return len(self.breadcrumbs) >= 1

    def update(self):
        context = self.context.aq_inner

        self.portal_state = getMultiAdapter((context, self.request), name="plone_
        ↪ portal_state")
        self.site_url = self.portal_state.portal_url()
        self.navigation_root_url = self.portal_state.navigation_root_url()

        breadcrumbs_view = getMultiAdapter((context, self.request), name='breadcrumbs_
        ↪ view')
        self.breadcrumbs = breadcrumbs_view.breadcrumbs()

        # right-to-left reading order
        self.is_rtl = self.portal_state.is_rtl()
```

Template code `templates/breadcrumbs.pt`:

```
<div id="portal-breadcrumbs"
    i18n:domain="plone"
    tal:condition="viewlet/visible"
    tal:define="breadcrumbs viewlet/breadcrumbs;
                is_rtl viewlet/is_rtl">

    <span id="breadcrumbs-home">
        <a i18n:translate="tabs_home"
            tal:attributes="href viewlet/navigation_root_url">Home</a>
        <span tal:condition="breadcrumbs" class="breadcrumbSeparator">
            <tal:ltr condition="not: is_rtl">|</tal:ltr>
```

```
        <tal:rtl condition="is_rtl">|</tal:rtl>
    </span>
</span>
<span tal:repeat="crumb breadcrumbs"
      tal:attributes="dir python:is_rtl and 'rtl' or 'ltr';
                      id string:breadcrumbs-#{repeat/crumb/number}">
    <tal:item tal:define="is_last repeat/crumb/end;
                        url crumb/absolute_url;
                        title crumb/Title">

        <a href="#"
          tal:omit-tag="not: url"
          tal:condition="python:not is_last"
          tal:attributes="href url"
          tal:content="title">
            crumb
        </a>
        <span class="breadcrumbSeparator" tal:condition="not: is_last">
            <tal:ltr condition="not: is_rtl">|</tal:ltr>
            <tal:rtl condition="is_rtl">|</tal:rtl>
        </span>
        <span id="breadcrumbs-current"
          tal:condition="is_last"
          tal:content="title">crumb</span>
    </tal:item>
</span>
</div>
```

Back button

Below is an example how we have extracted information like the parent container and such from breadcrumbs.

Note: We need special dealing for “default view” of objects... that’s the canonical part.

```
class Back(grok.Viewlet):
    """ Back button
    """

    def update(self):
        context = aq_inner(self.context)

        context_helper = getMultiAdapter((context, self.request), name="plone_context_
↪state")

        portal_helper = getMultiAdapter((context, self.request), name="plone_portal_
↪state")

        canonical = context_helper.canonical_object()

        parent = aq_parent(canonical)

        breadcrumbs_view = getView(self.context, self.request, 'breadcrumbs_view')
        breadcrumbs = breadcrumbs_view.breadcrumbs()
```

```
if (len(breadcrumbs)==1):
    self.backTitle = _(u"Home")
else:
    if hasattr(parent, "Title"):
        self.backTitle = parent.Title()
    else:
        self.backTitle = _(u"Back")

    if hasattr(parent, "absolute_url"):
        self.backUrl = parent.absolute_url()
    else:
        self.backUrl = portal_helper.portal_url()

self.isHome = len(breadcrumbs)==0
```

More info

- <http://code.google.com/p/plonegomobile/source/browse/gomobiletheme.basic/trunk/gomobiletheme/basic/viewlets.py#281>

Sitemap protocol

Introduction

Sitemap is used to submit the site content to search engines.

- <http://www.google.com/webmasters/>

Plone sitemap

Plone supports basic sitemap out of the box.

- <https://github.com/plone/plone.app.layout/blob/master/plone/app/layout/sitemap/sitemap.py>

Customized sitemap

Example

- <https://plonegomobile.googlecode.com/svn/trunk/gomobile/gomobile.mobile/gomobile/mobile/browser/sitemap.py>

Enabling sitemap programmatically

For unit tests:

```
# Sitemap must be enabled from the settings to access the view
self.portal.portal_properties.site_properties.enable_sitemap = True
```

Discussion and comments

Description

How to control commenting and discussion in Plone programmatically

Introduction

`plone.app.discussion` provides basic in-site discussion support.

Disqus is a popular external `<iframe>` embed service used for commenting.

More info

- <http://packages.python.org/plone.app.discussion/>
- <https://pypi.python.org/pypi/plone.app.discussion>

Content type support

Enable discussion in *portal_types* for each content type It's the *Allow Discussion* checkbox.

Discussion shows up as `plone.comments` viewlet in `plone.app.layout.viewlets.interfaces.IBelowContent` viewlet manager.

Getting total comment count

Example:

```
def getDiscussionCount(self):
    try:
        # plone.app.discussion.conversation object
        # fetched via IConversation adapter
        conversation = IConversation(self.targetContent)
    except:
        return 0

    return conversation.total_comments
```

Contact forms

Introduction

Plone ships with a

- site contact form which is form-to-mail to the site administration email
- document comment form
- email this to friend form

Default address `/contact-info`.

Customizing site contact form

Contact form files are

- Products/CMFPlone/skins/plone_templates/contact-info.cpt
- Products/CMFPlone/skins/plone_templates/contact-info.cpt.metadata
- Products/CMFPlone/skins/plone_templates/site_feedback_template.pt
- Products/CMFPlone/skins/plone_formscripts/validate_feedback.vpy
- Products/CMFPlone/skins/plone_formscripts/send_feedback.cpy
- Products/CMFPlone/skins/plone_formscripts/send_feedback.cpy.metadata
- Products/CMFPlone/skins/plone_formscripts/send_feedback_site.cpy
- Products/CMFPlone/skins/plone_formscripts/send_feedback_site.cpy.metadata

Inspect the files to know which you need to change. Copy these files to skin layer folder (any folder under skins) in your add-on product.

Note: Different contact form is displayed for logged-in and anonymous users. Logged in user email is not asked, but one stored in member properties is used.

Example

Below is an example how to add “phone number” field for all *not logged in* users feedback form.

Add a new optional field to contact-info.cpt (language hardcoded):

```
<div class="field">

  <label for="phone_number">
    Puhelinnumero
  </label>

  <div class="formHelp">
    Puhelinnumero, mik&#xe4;li haluatte teihin oltavan yhteydess&#xe4; puhelimitse.
  </div>

  <input type="text"
    id="phone_number"
    name="phone_number"
    size="25"
    value=""
    tal:attributes="value request/phone_number|nothing"
  />

</div>
```

Refer this field in site_feedback_template.pt:

```
<div i18n:domain="plone"
  tal:omit-tag=""
  tal:define="utool nocall:here/portal_url;
              portal utool/getPortalObject;
              charset portal/email_charset|string:utf-8;
              dummy python:request.RESPONSE.setHeader('Content-Type', 'text/plain;;
  ↪charset=%s' % charset);"
>
```

```
<div i18n:translate="site_feedback_mailtemplate_body" tal:omit-tag="">

You are receiving this mail because <span i18n:name="fullname" tal:omit-tag=""
↳tal:content="options/sender_fullname|nothing" />
<span i18n:name="from_address" tal:omit-tag="" tal:content="options/sender_from_
↳address"/>
is sending feedback about the site administered by you at <span i18n:name="url"
↳tal:replace="options/url" />.
The message sent was:

<span i18n:name="message" tal:omit-tag="" tal:content="options/message | nothing" />

</div>
--
<span tal:replace="portal/email_from_name" />

</div>

Puhelinnumero: <span tal:content="request/phone_number|nothing" />
```

Note: As a crude hack we add new field to the very bottom of the email, as everything side `<div i18n:translate>` is replaced from translation catalogs.

Replacing the site contact form with a content object

Sometimes you want to turn off the builtin form in favour of a piece of content. For example you might want a PloneFormGen form that content editors can alter. Naming your content item `contact-info` works because Zope traversal will find your content item before the page template. However Plone won't allow a new piece of content to be named `contact-info` since that's a reserved identifier, so the trick is to rename it in the Management Interface from the Plone-generated `contact-info-1` back to `contact-info`.

This works for `accessibility-info` too.

If you have a PFG contact form at, say, `about/contact-us` and want to turn off the builtin `contact-info` form, use the rename trick to create a `contact-info` Link object at the site root that points to your new form. Through acquisition, even URLs like `events/contact-info` will successfully redirect to your custom form.

Queries, search and indexing

In plone, database index and search facilities are provided by `portal_catalog` tool. There are two distinct functions. Indexing: All searchable data is copied to the catalog when the object is indexed, to make object searchable and listable. Querying: Search keys are matched against the indexed catalog copies to return the indexed metadata of the object.

Catalogs

Description

A brief introduction to ZCatalogs, the Catalog Tool and what they're used for.

Why ZCatalogs

Plone uses the ZODB to store content in a very free-form manner with arbitrary hierarchy and a lot of flexibility in general. For some content use cases, however, it is very useful to treat content as more ordered, or tabular. This is where ZCatalog comes in.

Searching, for example, requires being able to query content on structured data such as dates or workflow states. Additionally, query results often need to be sorted based on structured data of some sort. When it comes to searching it is very valuable to treat our free-form persistent ZODB objects as if they were more tabular. ZCatalog indexes do exactly this.

Since the ZCatalog is in the business of treating content as tabular when it isn't necessarily so, it is very tolerant of any missing data or exceptions when indexing. For example, Plone includes “start” and “end” indexes to support querying events on their start and end dates. When a page is indexed, however, it doesn't have start or end dates. Since the ZCatalog is tolerant, it doesn't raise any exception when indexing the start or end dates on a page. Instead it simply doesn't include pages in those indexes. As such, it is appropriate to use indexes in the catalog to support querying or sorting when not all content provides the data indexed.

This manual is intended to be a brief start guide to ZCatalogs, specially aimed to tasks specific to Plone, and will not treat advanced ZCatalogs concepts in depth. If you want to learn more about ZCatalogs in the context of Zope, please refer to [The Zope Book](#), [Searching and Categorizing Content](#).

Quick start

Every ZCatalog is composed of indexes and metadata. Indexes are fields you can search by, and metadata are copies of the contents of certain fields which can be accessed without waking up the associated content object.

Most indexes are also metadata fields. For example, you can search objects by *Title* and then display the *Title* of each object found without fetching them, but note not all indexes need to be part of metadata.

When you search inside the catalog, what you get as a result is a list of elements known as brains. Brains have one attribute for each metadata field defined in the catalog, in addition to some methods to retrieve the underlying object and its location. Metadata values for each brain are saved in the metadata table of the catalog upon the (re)indexing of each object.

Brains are said to be lazy for two reasons; first, because they are only created ‘just in time’ as your code requests each result, and second, because retrieving a catalog brain doesn't wake up (or load) the objects themselves, avoiding a huge performance hit.

To see the ZCatalogs in action, use your favorite browser and open the Management Interface. You'll see an object in the root of your Plone site named *portal_catalog*. This is the Catalog Tool, a Plone tool based on ZCatalog created by default in every Plone site which indexes all the created content.

Open it and click the *Catalog* tab, at the top of the screen. There you can see the full list of currently indexed objects, filter them by path, and update and remove entries. If you click on any entry, a new tab (or window) will open showing the metadata and index values for the selected indexed object. Note that most fields are “duplicated” in the *Index Contents* and *Metadata Contents* tables, but its contents have different formats. This is because indexes are meant to search in, and metadata to retrieve certain attributes from the content object without waking it up.

Back to the management view of the Catalog Tool, if you click the *Indexes* or the *Metadata* tab you'll see the full list of currently available indexes and metadata fields with its types and more. There you can also add and remove indexes and metadata fields. If you're working in a test environment, you can use this manager view for playing with the catalog, but beware indexes and metadata are usually added through GenericSetup and not using the Management Interface.

Other catalogs

Besides, the main portal catalog, the site contains other catalogs.

- `uid_catalog` maintains object look up by Unique Identified (UID). UID is given to the object when it is created and it does not change even if the object is moved around the site.
- `reference_catalog` maintains inter-object references by object unique identified (UID). Archetypes's Reference-Field uses this catalog. The catalog contains indexes UID, relationship, sourceUID, targetId and targetUID.
- Add-on products may install their own catalogs which are optimized for specific purposes. For example, `beta-haus.emaillogin` creates `email_catalog` which is used to speed-up login by email process.

Manually indexing object to a catalog

The default content object `reindexObject()` is defined in `CMFCatalogAware` and will update the object data to `portal_catalog`.

If your code uses additional catalogs, you need to manually update cataloged values after the object has been modified.

Example:

```
# Update email_catalog which maintains loggable email addresses
email_catalog = self.portal.email_catalog
email_catalog.reindexObject(myuserobject)
```

Manually uncatalog object to a catalog

Sometimes is useful to uncatalog object.

code

```
### uncatalog object name id
>>> brains = catalog(getId=id)
>>> for brain in brains:
...     catalog.uncatalog_object(brain.getPath())
```

Rebuilding a catalog

Catalog rebuild means walking through all the objects on Plone site and adding them to the catalog. Rebuilding the catalog is very slow as the whole database must be read through. Reasons for you to do this in code could be

- Creating catalog after setting up objects in the unit tests
- Rebuilding after massive content migration

How to trigger rebuild:

```
portal_catalog = self.portal.portal_catalog
portal_catalog.clearFindAndRebuild()
```

Retrieving unique values from a catalog

Catalogs have a `uniqueValues` method associated with each index. There are times when you will need to get a list of all the values currently stored on a particular index. For example if you wanted the highest and lowest price you might first need to retrieve the values currently indexed for price. This example demonstrates how you can list all the unique values on an index named 'price'.

```
portal_catalog = self.portal.portal_catalog
portal_catalog.Indexes['price'].uniqueValues()
```

the result would be a listing of all the prices stored in the 'price' index:

```
(0, 100000, 120000, 200000, 220000, 13500000, 16000000, 25000000)
```

Minimal code for creating a new catalog

```
from zope.interface import Interface, implements
from zope.component import getUtility

from Acquisition import aq_inner
from Acquisition import aq_parent

from AccessControl import ClassSecurityInfo
from Globals import InitializeClass
from Products.CMFPlone.utils import base_hasattr
from Products.CMFPlone.utils import safe_callable
from Products.CMFCore.permissions import ManagePortal
from Products.CMFCore.utils import getToolByName
from Products.ZCatalog.ZCatalog import ZCatalog
from Products.CMFPlone.CatalogTool import CatalogTool


class IMyCatalog(Interface):
    """
    """

class MyCatalog(CatalogTool):
    """
    A specific launch catalog tool
    """

    implements(IMyCatalog)

    title = 'specific catalog'
    id = 'my_catalog'
    portal_type = meta_type = 'MyCatalog'
    plone_tool = 1

    security = ClassSecurityInfo()
    _properties=(
        {'id':'title', 'type': 'string', 'mode':'w'},)

    def __init__(self):
```

```
ZCatalog.__init__(self, self.id)

security.declarePublic('enumerateIndexes')
def enumerateIndexes(self):
    """Returns indexes used by catalog"""
    return (
        ('id', 'FieldIndex', ()),
        ('portal_type', 'FieldIndex', ()),
        ('path', 'ExtendedPathIndex', ('getPhysicalPath')),
        ('getCanonicalPath', 'ExtendedPathIndex', ('getCanonicalPath')),
        ('isArchived', 'FieldIndex', ()),
        ('is_trashed', 'FieldIndex', ()),
        ('is_obsolete', 'FieldIndex', ()),
        ('Language', 'FieldIndex', ()),
        ('review_state', 'FieldIndex', ()),
        ('allowedRolesAndUsers', 'DPLARAUIndex', ()),
    )

security.declarePublic('enumerateMetadata')
def enumerateMetadata(self):
    """Returns metadata used by catalog"""
    return (
        'Title',
        'getId',
        'UID',
        'review_state',
        'created',
        'modified',
    )

security.declareProtected(ManagePortal, 'clearFindAndRebuild')
def clearFindAndRebuild(self):
    """Empties catalog, then finds all contentish objects (i.e. objects
        with an indexObject method), and reindexes them.
        This may take a long time.
    """

    def indexObject(obj, path):
        self.reindexObject(obj)

    self.manage_catalogClear()

    portal = getToolByName(self, 'portal_url').getPortalObject()
    portal.ZopeFindAndApply(portal,
        """ put your meta_type here """,
        obj_metatypes=(),
        search_sub=True, apply_func=indexObject)

InitializeClass(MyCatalog)
```

Register a new catalog via portal_setup

In toolset.xml add this lines

```
<?xml version="1.0"?>
<tool-setup>

  <required tool_id="my_catalog"
            class="catalog.MyCatalog"/>

</tool-setup>
```

archetype_tool catalog map

archetype_tool maintains map between content types and catalogs which are interested int them. When object is modified through Archetypes mechanisms, Archetypes post change notification to all catalogs enlisted.

See *Catalogs* tab on archetype_tool in Management Interface.

Map an catalog for an new type

code

```
at = getToolByName(context, 'archetype_tool')
at.setCatalogsByType('MetaType', ['portal_catalog', 'mycatalog',])
```

Additional info

- [ZCatalog source code](#).
- <http://wyden.com/plone/basics/searching-the-catalog>

Indexes and metadata

Description

How to program your custom fields and data queries through portal_catalog.

What does indexing mean?

Indexing is the action to make object data search-able. Plone stores available indexes in the database. You can create them through-the-web and inspect existing indexes in portal_catalog on Index tab.

The Catalog Tool can be configured through the Management Interface or programatically in Python but current best practice in the CMF world is to use GenericSetup to configure it using the declarative *catalog.xml* file. The Generic-Setup profile for Plone, for example, uses the *CMFPlone/profiles/default/catalog.xml* XML data file to configure the Catalog Tool when a Plone site is created. It is fairly readable so taking a quick look through it can be very informative.

When using a GenericSetup extension profile to customize the Catalog Tool in your portal, you only need to include XML for the pieces of the catalog you are changing. To add an index for the Archetypes location field, as in the example below, a policy package could include the following *profiles/default/catalog.xml*:

```
<?xml version="1.0"?>
<object name="portal_catalog" meta_type="Plone Catalog Tool">
  <index name="location" meta_type="FieldIndex">
    <indexed_attr value="location"/>
  </index>
</object>
```

The GenericSetup import handler for the Catalog Tool also supports removing indexes from the catalog if present using the “remove” attribute of the *<index>* element. To remove the “start” and “end” indexes used for events, for example, a policy package could include the following *profiles/default/catalog.xml*:

```
<?xml version="1.0"?>
<object name="portal_catalog" meta_type="Plone Catalog Tool">
  <index name="start" remove="True" />
  <index name="end" remove="True" />
</object>
```

Warning

Care must be taken when setting up indexes with GenericSetup - if the import step for a *catalog.xml* is run a second time (for example when you reinstall the product), the indexes specified will be destroyed, losing all currently indexed entries, and then re-created fresh (and empty!). If you want to workaround this behaviour, you can either update the catalog afterwards or add the indexes yourself in Python code using a custom import handler.

For more info, see this setuphandler <https://github.com/plone/plone.app.event/blob/master/plone/app/event/setuphandlers.py> in *plone.app.event* or these discussions on more about this problem:

- <http://plone.293351.n2.nabble.com/How-to-import-catalog-xml-without-emptying-the-indexes-td2302709.html>
 - <https://mail.zope.org/pipermail/zope-cmf/2007-March/025664.html>
-

Viewing indexes and indexed data

Indexed data

You can do this through *portal_catalog* in the Management Interface.

- Click *portal_catalog* in the portal root
- Click *Catalog* tab
- Click any object

Indexes and metadata columns

Available indexes are stored in the database, not in Python code. To see what indexes your site has

- Click *portal_catalog* in the portal root
- Click *Indexes* and *Metadata* tabs

Creating an index

To perform queries on custom data, you need to add the corresponding index to `portal_catalog` first.

E.g. If your *Archetypes* content type has a field:

```

schema = [

    DateField("revisitDate",
        widget = atapi.DateWidget(
            label="Revisit date",
            description="When you are alarmed this content should be
↪revisited (one month beforehand this date)",
            schemata="revisit"
        ),
    ]

class MyContent(...):

    # This is automatically run-time generated function accessor method,
    # but could be any hand-written method as well
    # def getMyCustomValue(self):
    #     pass

```

You can add a new index which will *index* the value of this field, so you can make queries based on it later.

See more information about *accessor methods*.

Note: If you want to create an index for content type you do not control yourself or if you want to do some custom logic in your indexer, please see *Custom index method* below.

Creating an index through the web

This method is suitable during development time - you can create an index to your Plone database locally.

- Go to the Management Interface
- Click `portal_catalog`
- Click Indexes tab
- On top right corner, you have a drop down menu to add new indexes. Choose the index type you need to add.
 - Type: `FieldIndex`
 - Id: `getMyCustomValue`
 - Indexed attributes: `getMyCustomValue`

You can use Archetypes accessors methods directly as an indexed attribute. In example we use `getMyCustomValue` for AT field `customValue`.

The type of index you need depends on what kind queries you need to do on the data. E.g. direct value matching, ranged date queries, free text search, etc. need different kind of indexes.

- After this you can query `portal_catalog`:

```
my_brains = context.portal_catalog(getMyCustomValue=111)
for brain in my_brains:
    print brain["getMyCustomValue"]
```

Adding index using add-on product installer

You need to have your own add-on product which registers new indexes when the add-on installer is run. This is the recommended method for repeated installations.

You can create an index

- Using catalog.xml where XML is written by hand
- Create the index through the web and export catalog data from a development site using *portal_setup* tool *Export* functionality. The index is created through-the-web as above, XML is generated for you and you can fine tune the resulting XML before dropping it in to your add-on product.
- Create indexes in Python code of add-on custom import step.
- As a prerequisite, your add-on product must have *GenericSetup profile support*.

This way is repeatable: index gets created every time an add-on product is installed. It is more cumbersome, however.

Warning: There is a known issue of indexed data getting pruned when an add-on product is reinstalled. If you want to avoid this then you need to create new indexes in add-on installer custom setup step (Python code).

The example below is not safe for data prune on reinstall. This file is `profiles/default/catalog.xml`. It installs a new index called `revisit_date` of `DateIndex` type.

```
<?xml version="1.0"?>
<object name="portal_catalog" meta_type="Plone Catalog Tool">
  <index name="revisit_date" meta_type="DateIndex">
    <property name="index_naive_time_as_local">True</property>
  </index>
</object>
```

For more information see

- <http://maurits.vanrees.org/weblog/archive/2009/12/catalog>

Custom index methods

The `plone.indexer` package provides method to create custom indexing functions.

Sometimes you want to index “virtual” attributes of an object computed from existing ones, or just want to customize the way certain attributes are indexed, for example, saving only the 10 first characters of a field instead of its whole content.

To do so in an elegant and flexible way, Plone>=3.3 includes a new package, `plone.indexer`, which provides a series of primitives to delegate indexing operations to adapters.

Let’s say you have a content type providing the interface `IMyType`. To define an indexer for your type which takes the first 10 characters from the body text, just type (assuming the attribute’s name is ‘text’):


```
from plone.indexer.decorator import indexer

@indexer(IMyType)
def mytype_description(object, **kw):
    return object.text[:10]
```

Finally, register this factory function as a named adapter using ZCML. Assuming you've put the code above into a file named `indexers.py`:

```
<adapter name="description" factory=".indexers.mytype_description" />
```

And that's all! Easy, wasn't it?

Note you can omit the `for` attribute because you passed this to the `@indexer` decorator, and you can omit the `provides` attribute because the thing returned by the decorator is actually a class providing the required `IIndexer` interface.

To learn more about the `plone.indexer` package, read [its doctest](#).

For more info about how to create content types, refer to the [developing add-ons section](#). For older Archetypes content types, see the [Plone 4 documentention on Archetypes](#)

Important note: If you want to adapt an Archetypes content type like Event or News Item, take into account you will have to feed the `indexer` decorator with the Zope 3 interfaces defined in `Products.ATContentTypes.interface.*` files, not with the deprecated Zope 2 ones into the `Products.ATContentTypes.interfaces` file.

Creating a metadata column

The same rules and methods apply for metadata columns as creating index above. The difference with metadata is that

- It is not used for searching, only displaying the search result
- You store always a value copy as is

To create metadata colums in your `catalog.xml` add:

```
<?xml version="1.0"?>
<object name="portal_catalog" meta_type="Plone Catalog Tool">
  <!-- Add a new metadata column which will read from context.getSignificant()_
  <function -->
    <column value="getSignificant"/>
</object>
```

When indexing happens and how to reindex manually

Content indexing happens automatically if:

- The object is modified by the user using the standard edit forms
- `portal_catalog` rebuild is run (from *Advanced* tab)

You must call `reindexObject()` manually if you:

- Directly call object field mutators
- Otherwise directly change any object data

`reindexObject()` method takes the optional argument *idxs* which will list the changed indexes. If *idxs* is not given, all related indexes are updated even though they were not changed.

Example:

```
obj.setTitle('Foobar')
# update only the index associated with this change
obj.reindexObject(idxs=['Title'])
```

If you add a new index you need to run *Rebuild catalog* to get the existing values from content objects into the new index.

Also, if you modify security related parameters (permissions), you need to call `reindexObjectSecurity()`.

Check the thread [Best practices on reindexing the catalog](#) for more tips on how to reduce memory consumption and speed up the process.

Warning: Unit test warning: Usually Plone reindexes modified objects at the end of each request (each transaction). If you modify the object yourself you are responsible to notify related catalogs about the new object data.

Index types

Zope 2 product `PluginIndexes` defines various `portal_catalog` index types used by Plone.

- `FieldIndex` stores values as is
- `DateIndex` and `DateRangeIndex` store dates (Zope 2 `DateTime` objects) in searchable format. The latter provides ranged searches.
- `KeywordIndex` allows keyword-style look-ups (query term is matched against all the values of a stored list)
- `ZCTextIndex` is used for full text indexing
- `ExtendedPathIndex` is used for indexing content object locations.

Default Plone indexes and metadata columns

Some interesting indexes

- `start` and `end`: Calendar event timestamps, used to make up calendar portlet
- `sortable_title`: Title provided for sorting
- `portal_type`: Content type as it appears in `portal_types`
- `Type`: Translated, human readable, type of the content
- `path`: Where the object is (`getPhysicalPath` accessor method).
- `object_provides`: What interfaces and marker interfaces object has. `KeywordIndex` of interface full names.
- `is_default_page`: `is_default_page` is method in `CMFPlone/CatalogTool.py` handled by `plone.indexer`, so there is nothing like `object.is_default_page` and this method calls `ptool.isDefaultPage(obj)`

Some interesting columns

- `getRemoteURL`: Where to go when the object is clicked

- `getIcon`: Since Plone 5.0.2 - Boolean value which is set to `:guilabel:True`, when item has or is an image (used for showing thumbs in lists, portlets, etc.). Content type icons (aka portaltype-icons) (e.g.: for folder, document, news item etc.) are rendered as fontello fonts since Plone 5.0.
- `exclude_from_nav`: If `True` the object won't appear in sitemap, navigation tree
- `mime_type`: Since Plone 5.1: Mime type information for content items where applicable (file, image, custom types,...) e.g.: `text/plain`, `image/jpeg`, `application/pdf` ...

Custom sorting by title

`sortable_title` is type of `FieldIndex` (raw value) and normal `Title` index is type of searchable text.

`sortable_title` is generated from `Title` in `Products/CMFPlone/CatalogTool.py`.

You can override `sortable_title` by providing an indexer adapter with a specific interface of your content type.

Example `indexes.py`:

```
from plone.indexer import indexer

from xxx.researcher.interfaces import IResearcher

@indexer(IResearcher)
def sortable_title(obj):
    """
    Provide custom sorting title.

    This is used by various folder functions of Plone.
    This can differ from actual Title.
    """

    # Remember to handle None value if the object has not been edited yet
    first_name = obj.getFirst_name() or ""
    last_name = obj.getLast_name() or ""

    return last_name + " " + first_name
```

Related `configure.zcml`

```
<adapter factory=".indexes.sortable_title" name="sortable_title" />
```

Full-text searching

Plone provides special index called `SearchableText` which is used on the site full-text search. Your content types can override `SearchableText` index with custom method to populate this index with the text they want to go into full-text searching.

Below is an example of having `SearchableText` on a custom Archetypes content class. This class has some methods which are not part of AT schema and thus must be manually added to `SearchableText`

```
def SearchableText(self):
    """
    Override searchable text logic based on the requirements.

    This method constructs a text blob which contains all full-text
    searchable text for this content item.
```

```
This method is called by portal_catalog to populate its SearchableText index.
"""

# Test this by enable pdb here and run catalog rebuild in the Management Interface
# xxx

# Speed up string concatenation ops by using a buffer
entries = []

# plain text fields we index from ourself,
# a list of accessor methods of the class
plain_text_fields = ("Title", "Description")

# HTML fields we index from ourself
# a list of accessor methods of the class
html_fields = ("getSummary", "getBiography")

def read(accessor):
    """
    Call a class accessor method to give a value for certain Archetypes field.
    """
    try:
        value = accessor()
    except:
        value = ""

    if value is None:
        value = ""

    return value

# Concatenate plain text fields as is
for f in plain_text_fields:
    accessor = getattr(self, f)
    value = read(accessor)
    entries.append(value)

transforms = getToolByName(self, 'portal_transforms')

# Run HTML valued fields through text/plain conversion
for f in html_fields:
    accessor = getattr(self, f)
    value = read(accessor)

    if value != "":
        stream = transforms.convertTo('text/plain', value, mimetype='text/html')
        value = stream.getData()

    entries.append(value)

# Plone accessor methods assume utf-8
def convertToUTF8(text):
    if type(text) == unicode:
        return text.encode("utf-8")
    return text
```

```
entries = [ convertToUTF8(entry) for entry in entries ]

# Concatenate all strings to one text blob
return " ".join(entries)
```

Other

- http://toutpt.wordpress.com/2008/12/14/archetype_tool-queuecatalog-becareful-with-indexing-with-plones-portal_catalog/

Querying

Description

How to programmatically search and query content from a Plone site.

- *Introduction*
- *Accessing the portal_catalog tool*
- *Querying portal_catalog*
 - *Available indexes*
- *Brain result id*
- *Brain result path*
- *Brain object schema*
 - *Getting the underlying object, its path, and its URL from a brain*
 - *getObject() and unrestrictedSearchResults() permission checks*
 - *Counting value of a specific index*
- *Sorting and limiting the number of results*
- *Text format*
- *Accessing indexed data*
- *Dumping portal catalog content*
- *Bypassing query security check*
- *Bypassing language check*
- *Bypassing Expired content check*
- *None as query parameter*
- *Query by path*
 - *Searching for content within a folder*
- *Query multiple values*

- *Querying by interface*
 - *Caveats*
- *Query by content type*
- *Query published items*
- *Getting a random item*
- *Querying FieldIndexes by Range*
- *Querying by date*
- *Query by language*
- *Boolean queries (AdvancedQuery)*
- *Setting Up A New Style Query*
- *Accessing metadata*
- *Fuzzy search*
- *Unique values*
- *Performance*
- *Batching*
- *Walking through all content*
- *Other notes*

Introduction

Querying is the action to retrieve data from search indexes. In Plone's case this usually means querying content items using either the `plone.api.content.find` function or directly using the `portal_catalog` tool.

Plone uses the *portal_catalog* tool to perform most content-related queries. Other special catalogs, like `reference_catalog` for Archetypes, exist, for specialized and optimized queries.

Accessing the `portal_catalog` tool

Plone queries are performed using `portal_catalog` which is available as an persistent object at the site root.

The recommended way to get the tool is using `plone.api.portal_get_tool`:

```
from plone import api
portal_catalog = api.portal.get_tool('portal_catalog')
```

Another safe method is to use the `getToolByName` helper function:

```
from Products.CMFCore.utils import getToolByName
catalog = getToolByName(context, 'portal_catalog')
```

Given the `portal`-object (site) itself is available also possible is the direct attribute access:

```
# portal_catalog is defined in the site root
portal_catalog = site.portal_catalog
```

There is also a another way, using traversing. This is discouraged, as this includes extra processing overhead:

```
# Use magical Zope acquisition mechanism
portal_catalog = context.portal_catalog
```

And here the same in TAL template, also discouraged:

```
<div tal:define="portal_catalog context/portal_catalog" />
```

Querying portal_catalog

To search for something and get the resulting brains, write:

```
results = catalog.searchResults(**kwargs)
```

Note: The catalog returns “*brains*”. A brain is a lightweight proxy for a found object, which has attributes corresponding to the metadata defined for the catalog.

Where `kwargs` is a dictionary of index names and their associated query values. Only the indexes that you care about need to be included. This is really useful if you have variable searching criteria. For example, coming from a form where the users can select different fields to search for.

```
results = catalog.searchResults(**{'portal_type': 'Event', 'review_state': 'pending'})
```

It is worth pointing out at this point that the indexes that you include are treated as a logical AND, rather than OR. In other words, the query above will find all the items that are both an Event, AND in the review state of pending.

Additionally, you can call the catalog tool directly, which is exactly the same to calling `catalog.searchResults()`:

```
.. code-block:: python
```

```
results = catalog(portal_type='Event')
```

If you call `portal_catalog()` without arguments it will return all indexed content objects:

```
.. code-block:: python
```

```
# Print all content on the site all_brains = catalog() for brain in all_brains:
```

```
    print('Name: ' + brain.Title + ', URL:' + brain.getURL())
```

The catalog tool queries return an iterable of catalog brain objects.

As mentioned previously, brains contain a subset of the actual content object information. The available subset is defined by the metadata columns in `portal_catalog`. You can see available metadata columns on the `portal_catalog` “Metadata” tab in Management Interface. For more information, see [indexing](#).

Available indexes

To see the full list of available indexes in your catalog

- open the Management Interface (which usually means navigating to `http://yoursiteURL/manage`)
- look for the `portal_catalog` object tool in the root of your Plone site and

- check the *Indexes* tab.

Note that there are different types of indexes, and each one admits different types of search parameters, and behaves differently. For example, *FieldIndex* and *KeywordIndex* support sorting, but *ZCTextIndex* doesn't. To learn more about indexes, see [The Zope Book, Searching and Categorizing Content](#).

Some of the most commonly used ones are:

Title The title of the content object.

Description The description field of the content.

Subject The keywords used to categorize the content. Example:

```
catalog.searchResults(Subject=('cats', 'dogs'))
```

portal_type As its name suggests, search for content whose portal type is indicated. For example:

```
catalog.searchResults(portal_type='News Item')
```

You can also specify several types using a list or tuple format:

```
catalog.searchResults(portal_type=('News Item', 'Event'))
```

review_state The current workflow review state of the content. For example:

```
catalog.searchResults(review_state='pending')
```

created, last_modified, effective, expires, start, end The dates stored with the content (“start” and “end” only on events). Example to find all content expired before now

```
import datetime

catalog.searchResults(
    created={'expired': datetime.datetime.now(), range='max'})
}
```

object_provides You can search by the interface provided by the content. Example:

```
from Products.MyProduct.path.to import IIsCauseForCelebration
catalog(object_provides=IIsCauseForCelebration.__identifier__)
```

Searching for interfaces can have some benefits. Suppose you have several types, for example, event types like *Birthday*, *Wedding* and *Graduation*, in your portal which implement the same interface (for example, *IIsCauseForCelebration*). Such an interface can be an *Dexterity* behavior (the behavior itself or its marker). Suppose you want to get items of these types from the catalog by their interface. This is more exact and more flexible than naming the types explicitly (like `portal_type=['Birthday','Wedding','Graduation']`), because you don't really care what the types' names really are: all you really care for is the interface. This has the additional advantage that if products added or modified later add types which implement the interface, these new types will also show up in your query.

Brain result id

Result ID (RID) is given with the brain object and you can use this ID to query further info about the object from the catalog.

Example:


```
(Pdb) brain.getRID()  
872272330
```

Brain result path

Brain result path can be extracted as string using `getPath()` method:

```
print(r.getPath())  
/site/sisalto/ajankohtaista
```

Brain object schema

To see what metadata columns a brain object contain, you can access this information from `__record_schema__` attribute which is a dict.

Example:

```
for i in brain.__record_schema__.items():  
    print(i)  
  
( 'startDate', 32 )  
( 'endDate', 33 )  
( 'Title', 8 )  
( 'color', 31 )  
( 'data_record_score_', 35 )  
( 'exclude_from_nav', 13 )  
( 'Type', 9 )  
( 'id', 19 )  
( 'cmf_uid', 29 )
```

Getting the underlying object, its path, and its URL from a brain

Searching inside the catalog returns catalog brains, not the object themselves. If you want to get the object associated with a brain, do:

```
brain.getObject()
```

To get the path of the object without fetching it:

which returns the path as a string, corresponding to `obj.getPhysicalPath()`

And finally, to get the URL of the underlying object, usually to provide a link to it:

```
brain.getURL()
```

which is equivalent to `obj.absolute_url()`.

Note: Calling `getObject()` has performance implications. Waking up each object needs a separate query to the database.

getObject() and unrestrictedSearchResults() permission checks

You cannot call `getObject()` for a restricted result, even in trusted code.

Instead, you need to use:

```
unrestrictedTraverse(brain.getPath())
```

For more information, see

- <http://www.mail-archive.com/zope-dev@zope.org/msg17514.html>

Counting value of a specific index

The efficient way of counting the number value of an index is to work directly in this index. For example we want to count the number of each `portal_type`. Querying via search results is a performance bottleneck for that. Iterating on all brains put those in zodb cache. This method is also a memory bottleneck.

A good way to achieve this would be:

```
# count portal_type index
stats = {}
x = getToolByName(context, 'portal_catalog')
index = x._catalog.indexes['portal_type']
for key in index.uniqueValues():
    t = index._index.get(key)
    if type(t) is not int:
        stats[str(key)] = len(t)
    else:
        stats[str(key)] = 1
```

Sorting and limiting the number of results

To sort the results, use the `sort_on` and `sort_order` arguments. The `sort_on` argument accepts any available index, even if you're not searching by it. The `sort_order` can be either 'ascending' or 'descending', where 'ascending' means from A to Z for a text field. 'reverse' is an alias equivalent to 'descending'.

```
results = catalog_searchResults(
    Description='Plone documentation',
    sort_on='sortable_title',
    sort_order='ascending'
)
```

It is possible to order to sort first in order of `portal_type` and second for the same types in order of `sortable_title`.

```
results = catalog_searchResults(
    Description='Plone documentation',
    sort_on='portal_type, sortable_title',
    sort_order='ascending'
)
```

Note: If you sort on something, the result will not contain items which aren't in the sort index. I.e. if you sort on `start` only items will be found having a `start` date, like events.

The `catalog.searchResults()` returns a list-like object, so to limit the number of results you can just use Python's slicing. For example, to get only the first 3 items:

```
results = catalog.searchResults(Description='Plone documentation')[:3]
```

In addition, ZCatalogs allow a `sort_limit` argument. The `sort_limit` is only a hint for the search algorithms and can potentially return a few more items, so it's preferable to use both `sort_limit` and slicing simultaneously:

```
limit = 50
results = catalog.searchResults(
    Description='Plone documentation',
    sort_limit=limit
)[:limit]
```

`portal_catalog` query takes `sort_on` argument which tells the index used for sorting. `sort_order` defines sort direction. It can be string "reverse".

Sorting is supported only on FieldIndexes and some derived indexes. Due to the nature of searchable text indexes (they index split text, not strings) they cannot be used for sorting. For example, to do sorting by title, an index called `sortable_title` should be used.

Example of how to sort by id:

```
results = context.portal_catalog.searchResults(
    sort_on='id',
    portal_type='Document',
    sort_order='reverse'
)
```

Text format

Indexes use direct attribute access (Dexterity) and so return the raw value. This depends on the schema and is i.e. for a `TextLine` unicode.

For some indexes special index-adapters are registered. Here it is upon the indexer implementation how the value is returned.

With Archetypes, accessors are used to index the field value and the returned text is UTF-8 encoded. This is a limitation inherited from the early ages of Plone. To get unicode value for e.g. title you need to do the following:

```
title = brain['Title']
title = title.decode('utf-8')
```

Accessing indexed data

Normally you don't get copy of indexed data with brains, only metadata. You can still access the raw indexed data if you know what you are doing by using RID of the brain object.

Note: This is a very rare use case and documented here for completeness.

Example:

```
(Pdb) data = self.context.portal_catalog.getIndexDataForRID(872272330)
(Pdb) for i in data.items(): print i
('Title', ['ulkomuseon', 'tarinaopastukset'])
('effectiveRange', (21305115, 278752140))
('object_provides', ['Products.CMFCore.interfaces._content.IDublinCore', 'Products.
→ATContentTypes.interface.interfaces.IHistoryAware', 'AccessControl.interfaces.IOwned
→', 'OFS.interfaces.ITraversable', 'plone.portlets.interfaces.ILocalPortletAssignable
→', 'Products.Archetypes.interfaces._base.IBaseObject', 'zope.annotation.interfaces.
→IAttributeAnnotatable', 'vs.event.interfaces.IVSEvent', 'Products.CMFCore.
→interfaces._content.IMutableMinimalDublinCore', 'OFS.interfaces.IPropertyManager',
→'OFS.interfaces.IZopeObject', 'AccessControl.interfaces.IRoleManager', 'zope.
→annotation.interfaces.IAnnotatable', 'Acquisition.interfaces.IAcquirer', 'Products.
→ATContentTypes.interface.event.IATEvent', 'OFS.interfaces.ICopySource', 'Products.
→ATContentTypes.interface.interfaces.ICalendarSupport', 'Products.ATContentTypes.
→interface.interfaces.IATContentType', 'plone.app.iterate.interfaces.IIterateAware',
→'Products.Archetypes.interfaces._base.IBaseContent', 'Products.CMFCore.interfaces._
→content.ICatalogableDublinCore', 'Products.CMFDynamicViewFTI.interface._base.
→IBrowserDefault', 'Products.Archetypes.interfaces._referenceable.IReferenceable',
→'plone.locking.interfaces.ITTWLockable', 'plone.app.imaging.interfaces.IBaseObject',
→'persistent.interfaces.IPersistent', 'webdav.interfaces.IDAVResource',
→'AccessControl.interfaces.IPermissionMappingSupport', 'OFS.interfaces.ISimpleItem',
→'plone.app.kss.interfaces.IPortalObject', 'plone.app.kss.interfaces.IContentish',
→'archetypes.schemaextender.interfaces.IExtensible', 'App.interfaces.IUndoSupport',
→'OFS.interfaces.IManageable', 'App.interfaces.IPersistentExtra', 'Products.CMFCore.
→interfaces._content.IMutableDublinCore', 'Products.Archetypes.interfaces._
→athistoryaware.IATHistoryAware', 'dateable.kalends.IRecurringEvent', 'OFS.
→interfaces.IItem', 'zope.interface.Interface', 'OFS.interfaces.IFTPAccess',
→'Products.CMFDynamicViewFTI.interface._base.ISelectableBrowserDefault', 'webdav.
→interfaces.IWriteLock', 'Products.CMFCore.interfaces._content.IMinimalDublinCore',
→'Products.CMFCore.interfaces._content.IDynamicType', 'Products.CMFCore.interfaces._
→content.IContentish'])
('Type', u'VSEvent')
('id', 'ulkomuseon-tarinaopastukset')
('cmf_uid', 2)
('recurrence_days', [733960, 733981, 733974, 733967])
('end', 1077028380)
('Description', ['saamelaismuseon', 'ulkomuseossa', ...
('is_folderish', False)
('getId', 'ulkomuseon-tarinaopastukset')
('start', 1077028380)
('is_default_page', False)
('Date', 1077036795)
('review_state', 'published')
('Language', <LanguageIndex.IndexEntry id 872272330 language fi, cid_
→8b9a08c216b8e086f3446775ad71a748>)
('portal_type', 'VSEvent')
('expires', 1339244460)
('allowedRolesAndUsers', ['Anonymous'])
('getObjPositionInParent', 10)
('path', '/siida/sisalto/8-vuodenaikaa/ulkomuseon-tarinaopastukset')
('in_reply_to', '')
('UID', '8b9a08c216b8e086f3446775ad71a748')
('Creator', 'admin')
('effective', 1077036795)
('getRawRelatedItems', [])
('getEventType', [])
('created', 1077036792)
```

```
( 'modified', 1077048720)
( 'SearchableText', ['ulkomuseon', 'tarinaopastukset', ...
( 'sortable_title', 'ulkomuseon tarinaopastukset')
( 'meta_type', 'VSEvent')
( 'Subject', [])
```

You can also directly access a single index:

```
# Get event brain result id
rid = event.getRID()
# Get list of recurrence_days indexed value.
# ZCatalog holds internal Catalog object which we can directly poke in evil way
# This call goes to Products.PluginIndexes.UnIndex.Unindex class and we
# read the persistent value from there what it has stored in our index
# recurrence_days
index = portal_catalog._catalog.getIndex('recurrence_days')
indexed_days = index.getEntryForObject(rid, default=[])
```

Dumping portal catalog content

Following is useful in unit test debugging.

```
# Print all objects visible to the currently logged in user
for i in portal_catalog(): print i.getURL()
```

Bypassing query security check

Note: Security: All portal_catalog queries are limited to the current user permissions by default.

If you want to bypass this restriction, use the unrestrictedSearchResults() method.

```
# Print absolute content of portal_catalog
for i in portal_catalog.unrestrictedSearchResults():
    print i.getURL()
```

With unrestrictedSearchResults() you need also a special way to get access to the objects without triggering a security exception:

```
.. code-block:: python
```

```
obj = brain._unrestrictedGetObject()
```

Bypassing language check

Note: All portal_catalog() queries are limited to the selected language of the current user. You need to explicitly bypass the language check if you want to do multilingual queries.

Language is only a factor when a multilingual product is installed - which basically comes down to one of the venerable *LinguaPlone* or the more modern *plone.app.multilingual*. Bypassing the language check depends on which of these you are using.

In *LinguaPlone* and *plone.app.multilingual 1.x* (what you would probably use in versions 4.3 or earlier of Plone), a patch is applied to the `portal_catalog`. To bypass this add the parameter `Language='all'` to your catalog query like so:

```
all_content_brains = portal_catalog(Language='all')
```

plone.app.multilingual 2.x and later (part of Plone 5.x) creates Root Language Folders for each of your site's languages. It keeps ("jails") content within the appropriate folders. Each Root Language Folder is also a `NavigationRoot`, so the `portal_catalog` is already effectively limited to searches in the users current language. This means that the way to bypass this is to add the query parameter `path='/'` to your catalog query like so:

```
all_content_brains = portal_catalog(path='/')
```

Note: Although in *LinguaPlone* eventually the language folders are also marked to be an `INavigationRoot`. The language of the content is not enforced inside the language folder. In *plone.app.multilingual* there's a subscriber that moves the content to the appropriate folder.

Bypassing Expired content check

Plone and its `portal_catalog` have a mechanism to list only active (non-expired) content by default.

Below is an example of how the expired content check is made:

```
.. code-block:: python
```

```
mtool = context.portal_membership show_inactive = mtool.checkPermission('Access inactive portal content', context)
```

```
contents = context.portal_catalog.queryCatalog(show_inactive=show_inactive)
```

See also:

- *Listing*

None as query parameter

Warning: Usually if you pass in `None` as the query value, it will match all the objects instead of zero objects.

Note: Querying for `None` values is possible with [AdvancedQuery](#) (see below).

Query by path

[ExtendedPathIndex](#) is the index used for content object paths. The *path* index stores the physical path of the objects.

Warning: If you ever rename your Plone site instance, the path index needs to be completely rebuilt.

Example, return myfolder and all child content.

```
portal_catalog(path={ "query": "/myploneinstance/myfolder" })
```

Searching for content within a folder

Use the ‘path’ argument to specify the physical path to the folder you want to search into.

By default, this will match objects into the specified folder and all existing sub-folders. To change this behaviour, pass a dictionary with the keys ‘query’ and ‘depth’ to the ‘path’ argument, where

- ‘query’ is the physical path, and
- ‘depth’ can be either 0, which will return only the brain for the path queried against, or some number greater, which will query all items down to that depth (eg, 1 means searching just inside the specified folder, or 2, which means searching inside the folder, and inside all child folders, etc).

The most common use case is listing the contents of an existing folder, which we’ll assume to be the `context` object in this example:

```
folder_path = '/'.join(context.getPhysicalPath())
results = catalog(path={'query': folder_path, 'depth': 1})
```

The above can be achieved much easier using `plone.api`:

```
from plone import api
results = api.content.find(context=context, depth=1)
```

Query multiple values

`KeywordIndex` index type indexes lists of values. It is used e.g. by Plone’s categories (subject) feature and `object_provides` provided interfaces index.

You can either query

- a single value in the list
- many values in the list (all must present)
- any value in the list

The index of the catalog to query is either the name of the keyword argument, a key in a mapping, or an attribute of a record object.

Attributes of record objects

- `query` – either a sequence of objects or a single value to be passed as query to the index (mandatory)
- `operator` – specifies the combination of search results when query is a sequence of values. (optional, default: ‘or’). Allowed values: ‘and’, ‘or’

Below is an example of matching any of multiple values gives as a Python list in `KeywordIndex`. It queries all event types and `recurrence_days` `KeywordIndex` must match any of the given dates:

```
.. code-block:: python
```

```
# Query all events on the site # Note that there is no separate list for recurrent events # so if you
# want to speed up you can hardcode # recurrent event type list here. matched_recurrence_events =
self.context.portal_catalog(

    portal_type=supported_event_types, recurrence_days={

        'query':recurrence_days_in_this_month, 'operator' : 'or'

    }

)
```

Querying by interface

Suppose you have several content types (for example, event types like 'Birthday','Wedding','Graduation') in your portal which implement the same interface (for example, `IIsCauseForCelebration`). Suppose you want to get items of these types from the catalog by their interface. This is more exact than naming the types explicitly (like `portal_type=['Birthday', 'Wedding', 'Graduation']`), because you don't really care what the types' names really are: all you really care for is the interface.

This has the additional advantage that if products added or modified later add types which implement the interface, these new types will also show up in your query.

Import the interface:

```
from Products.MyProduct.interfaces import IIsCauseForCelebration
catalog(object_provides=IIsCauseForCelebration.__identifier__)
```

In a script, where you can't import the interface due to restricted Python, you might do this:

```
object_provides='Products.MyProduct.interfaces.IIsCauseForCelebration'
```

The advantage of using `.__identifier__` instead of a dotted name-string is that you will get errors at startup time if the interface cannot be found. This will catch typos and missing imports.

Caveats

- `object_provides` is a `KeywordIndex` which indexes absolute Python class names. A string matching is performed for the dotted name. Thus, you will have zero results for this:

```
catalog(object_provides="Products.ATContentTypes.interface.IATDocument")
```

because `Products.ATContentTypes.interface` imports everything from `document.py`. But this will work:

```
catalog(object_provides="Products.ATContentTypes.interface.document.IATDocument")
# products.atcontenttypes.document.iatdocument declares the interfacea
```

- As with all catalog queries, if you pass an empty value for search parameter, it will return all results. so if the interface you defined would yield a none type object, the search would return all values of `object_provides`.

(Originally from [this tutorial](#).)

Note: Looks like query by `Products.CMFCore.interfaces._content.IFolderish` does not seem to work in Plone 4.1 as this implementation information is not populated in `portal_catalog`.

Query by content type

To get all catalog brains of certain content type on the whole site:

```
campaign_brains = self.context.portal_catalog(portal_type="News Item")
```

To see available type names, visit `portal_types` in the Management Interface.

Query published items

By default, the `portal_catalog` query does not care about the workflow state. You might want to limit the query to published items.

Example:

```
campaign_brains = self.context.portal_catalog(portal_type="News Item", review_state=
↪ "published")
```

`review_state` is a `portal_catalog` index which reads `portal_workflow` variable “`review_state`”. For more information, see what `portal_workflow` tool *Content* tab in Management Interface contains.

Getting a random item

The following view snippet allows you to get one random item on the site:

```
import random

def getRandomCampaign(self):
    """
    """

    campaign_brains = self.context.portal_catalog(portal_type="CampaignPage", review_
↪ state="published")

    # Filter out the current item which we have

    bad_ids = [ "you", "might", "want to black list some ids here" ]

    items = [ brain for brain in campaign_brains if brain["getId"] not in bad_ids ]

    # Check that we have items left after filtering

    items = list(items)

    if len(items) >= 1:
        # Pick one
        chosen = random.choice(items)
        return chosen.getObject()
```

```
else:
    # Fallback to the current content item if no random options available
    return self.context
```

Querying FieldIndexes by Range

The following examples demonstrate how to do range based queries. This is useful if you want to find the “minimum” or “maximum” values of something, the example assumes that there is an index called ‘getPrice’.

Get a value that is greater than or equal to 2:

```
items = portal_catalog({'getPrice':{'query':2,'range':'min'}})
```

Get a value that is less than or equal to 40:

```
items = portal_catalog({'getPrice':{'query':40,'range':'max'}})
```

Get a value that falls between 2 and 1000:

```
items = portal_catalog({'getPrice':{'query':[2,1000],'range':'min:max'}})
```

Querying by date

See [DateIndex](#).

Example:

```
date_range = {
    'query': (
        DateTime('2002-05-08 15:16:17'),
        DateTime('2062-05-08 15:16:17'),
    ),
    'range': 'min:max',
}

items = portal_catalog(effective=date_range)
```

Note that `effectiveRange` may be a lot more efficient. This will return only objects whose `effective_date` is in the past, ie. objects that are not unpublished:

```
items = portal_catalog(effectiveRange=DateTime())
```

Example 2 - how to get items one day old of `FeedFeederItem` content type:

```
# DateTime deltas are days as floating points
end = DateTime.DateTime() + 0.1 # If we have some clock skew peek a little to the_
↪future
start = DateTime.DateTime() - 1

date_range_query = { 'query':(start,end), 'range': 'min:max' }

items = portal_catalog.queryCatalog({"portal_type":"FeedFeederItem",
                                     "created" : date_range_query,
                                     "sort_on":"positive_ratings",
                                     "sort_order":"reverse",
```

```
"sort_limit":count,
"review_state":"published"))
```

Example 3: how to get news items for a particular year in the template code

```
<div metal:fill-slot="main" id="content-news"
tal:define="boundLanguages here/portal_languages/getLanguageBindings;
prefLang python:boundLanguages[0];
DateTime python:modules['DateTime'].DateTime;
start_year request/year| python: 2004;
end_year request/year| python: 2009;
start_year python: int(start_year);
end_year python: int(end_year);
results python:container.portal_catalog(
    portal_type='News Item',
    sort_on='Date',
    sort_order='reverse',
    review_state='published',
    id=prefLang,
    created= ( 'query' : [DateTime(start_year,1,1), DateTime(end_year,12,
↪31)], 'range':'minmax' )
);
results python:[r for r in results if r.getObject()];
Batch python:modules['Products.CMFPlone'].Batch;
b_start python:request.get('b_start',0);
portal_discussion nocall:here/portal_discussion;
isDiscussionAllowedFor nocall:portal_discussion/isDiscussionAllowedFor;
getDiscussionFor nocall:portal_discussion/getDiscussionFor;
home_url python: mtool.getHomeUrl;
localized_time python: modules['Products.CMFPlone.PloneUtilities'].
↪localized_time;">
...
</div>
```

Example 4 - how to get upcoming events of next two months:

```
def formatDate(self, event):
    """
    """
    dt = event["start"]
    return dt.strftime("%d.%m.%Y")

def update(self):
    portal_catalog = self.context.portal_catalog

    start = DateTime.DateTime() - 1 # yesterday
    end = DateTime.DateTime() + 60 # Two months future
    date_range_query = {'query': (start, end), 'range': 'min:max'}

    count = 5

    self.events = portal_catalog.queryCatalog({"portal_type": "Event",
                                                "start": date_range_query,
                                                "sort_on": "start",
                                                "sort_order": "reverse",
                                                "sort_limit": count,
                                                "review_state": "published"})
```

More info

- http://www.ifpeople.net/fairsource/courses/material/apiPlone_en

Query by language

You can query by language:

```
portal_catalog({'Language':"en"})
```

Note: plone.app.multilingual must be installed.

Boolean queries (AdvancedQuery)

AdvancedQuery is an add-on product for Zope's ZCatalog providing queries using boolean logic. AdvancedQuery is developer level product, providing Python interface for constructing boolean queries.

AdvancedQuery monkey-patches portal_catalog to provide new method portal_catalog.evalAdvancedQuery().

Example:

```
from Products import AdvancedQuery

portal_catalog = self.portal_catalog # Acquire portal_catalog from higher hierarchy_
↳level

path = self.getPhysicalPath() # Limit the search to the current folder and its_
↳children

# object.getPhysicalPath() returns the path as tuples of path parts
# Convert path to string
path = "/".join(path)

# Limit search to path in the current contex object and
# match all children implementing either of two interfaces
# AdvancedQuery operations can be combined using Python expressions & | and ~
# or AdvancedQuery objects
query = AdvancedQuery.Eq("path", path) & (AdvancedQuery.Eq("getMyIndexGetter1", "foo
↳") | AdvancedQuery.Eq("getMyIndexGetter2", "bar"))

# The following result variable contains iterable of CatalogBrain objects
results = portal_catalog.evalAdvancedQuery(query)

# Convert the catalog brains to a Python list containing tuples of object unique ID_
↳and Title
pairs = []
for nc in results:
    pairs.append((nc["UID"], nc["Title"]))

# query = Eq("path", diagnose_path) & Eq("SearchableText", text_query_target)
query = Eq("path", diagnose_path) & Eq("SearchableText", text_query_target)
```

```
return self.context.portal_catalog.evalAdvancedQuery(query)
```

Note: Plone 3 ships with AdvancedQuery but it is not part of Plone. Always declare AdvancedQuery dependency in your egg's setup.py install_requires.

Warning: AdvancedQuery does not necessarily apply the same automatic limitations which normal portal_catalog() queries do, like language and expiration date. Always check your query code against these limitations.

More information

- See `AdvancedQuery`.
- <https://plone.org/documentation/manual/upgrade-guide/version/upgrading-plone-3-x-to-4.0/updating-add-on-products-for-plone-4.0/removed-advanced-query>

Setting Up A New Style Query

With Plone 4.2, collections use so-called new-style queries by default. These are, technically speaking, canned queries, and they appear to have the following advantages over old-style collection's criteria:

- They are not complicated sub-objects of collections, but comparably simple subobjects that can be set using simple Python expressions.
- These queries are apparently much faster to execute, as well as
- much easier to understand, and
- content type agnostic in the sense that they are no longer tied to ArcheTypes.

The easiest way to get into these queries is to grab a debug shell alongside an instance, then fire up a browser pointing to that instance, then manipulate the queries and watch the changes on the debug shell, if you want to experiment. I've constructed a dummy collection for demonstration purposes, named *testquery*. I've formatted the output a little, for readability.

Discovering the query:

```
>>> site.invokeFactory('Collection', id='testquery') # actually with my browser
>>> tq = site['testquery']
>>> tq.getRawQuery()
[
  {'i': 'created', 'o': 'plone.app.querystring.operation.date.today'},
  {'i': 'Description', 'o': 'plone.app.querystring.operation.string.contains', 'v':
↪ 'my querystring'},
  {'i': 'portal_type', 'o': 'plone.app.querystring.operation.selection.is', 'v': [
↪ 'Document']},
  {'i': 'Subject', 'o': 'plone.app.querystring.operation.selection.is', 'v': ['some_
↪ tag']}
]
>>> tq.getSort_on()
'effective'
>>> tq.getSort_reversed()
True
>>> tq.getLimit()
```

```
1000
>>> tq.selectedViewFields()
[
    ('Title', u'Title'),
    ('Creator', 'Creator'),
    ('Type', u'Item Type'),
    ('ModificationDate', u'Modification Date'),
    ('ExpirationDate', u'Expiration Date'),
    ('getId', u'Short Name'),
    ('getObjSize', u'Size')
]
```

This output should be pretty self-explaining: This query finds objects that were created today, which have “my querystring” in their description, are of type “Document” (ie, “Page”), and have “some_tag” in their tag set (you’ll find that under “Classification”). Also, the results are being sorted in reverse order of the Effective Date (ie, the publishing date). We’re getting at most 1000 results, which is the default cut-off.

You can set the query expression (individual parts are evaluated as logical AND) using

```
>>> tq.setQuery( your query expression, see above )
```

The three parts of an individual query term are

- ‘i’: which index to query
- ‘o’: which operator to use (see *plone.app.querystring* for a list)
- ‘v’: the possible value of an argument to said operator - eg. the query string.

Other parameters can be manipulated the same way:

```
>>> tq.setSort_reversed(True)
```

Accessing metadata

Metadata is collected from the object during cataloging and is copied to brain object for faster access (no need to wake up the actual object from the database).

ZCatalog brain objects use Python dictionary-like API to access metadata. Below is a fail-safe example for a metadata access:

```
def getImageTag(self, brain):
    """
    Get lead image for ZCatalog brain in folder listing.

    (Based on collective.contentleadimage add-on product)

    @param brain: Products.ZCatalog.Catalog.mybrains object

    @return: HTML source code for content lead <img>
    """

    # First check if the index exist
    if not brain.has_key("hasContentLeadImage"):
        return None

    # Index can have indexed value None or
    # custom value Missing.Value if the indexer
```

```

# for brain's object failed to run or returned Missing.
# Both of these values evaluate to False in Python
has_image = brain["hasContentLeadImage"]

# The value was missing, None or False
if not has_image:
    return None

context = brain.getObject()

# AT inspection API
field = context.getField(IMAGE_FIELD_NAME)
if not field:
    return None

# ImageField.tag() API
if field.get_size(context) != 0:
    scale = "tile" # 64x64
    return field.tag(context, scale=scale)

```

Note: This is for example purposes only - the code above is working, but not optimal, and can be written up without waking up the object.

Fuzzy search

- <https://pypi.python.org/pypi/c2.search.fuzzy/>

Unique values

ZCatalog has *uniqueValuesFor()* method to retrieve all unique values for a certain index. It is intended to work on FieldIndexes only.

Example:

```

# getArea() is Archetype accessor for area field
# which is a string and tells the content area.
# Custom getArea FieldIndex indexes these values
# to portal catalog.
# The following line gives all area values
# inputted on the site.
areas = portal_catalog.uniqueValuesFor("getArea")

```

Performance

The following community mailing list blog posts is very insightful about the performance characteristics of Plone search and indexing:

- <http://plone.293351.n2.nabble.com/Advice-for-site-with-very-large-number-of-objects-millions-tp5513207p5529103.html>

Batching

Example:

```
results = Batch(contents, self.b_size, self.b_start, orphan=0)
```

- orphan - the next page will be combined with the current page if it does not contain more than orphan elements

Walking through all content

`portal_catalog()` call without search parameters will return all indexed site objects.

Here is an example how to crawl through Plone content to search HTML snippets. This can be done by rendering every content object and check whether certain substrings exists the output HTML This snippet can be executed through-the-web in the Management Interface.

This kind of scripting is especially useful if you need to find old links or migrate some text / HTML snippets in the content itself. There might be artifacts which only appear on the resulting pages (portlets, footer texts, etc.) and thus they are invisible to the normal full text search.

Example:

```
# Find arbitrary HTML snippets on Plone content pages

# Collect script output as text/html, so that you can
# call this script conveniently by just typing its URL to a web browser
buffer = ""

# We need to walk through all the content, as the
# links might not be indexed in any search catalog
for brain in context.portal_catalog(): # This queries cataloged brain of every
    ↪content object
    try:
        obj = brain.getObject()
        # Call to the content object will render its default view and return it as
        ↪text
        # Note: this will be slow - it equals to load every page from your Plone site
        rendered = obj()
        if "yourtextmatch" in rendered:
            # found old link in the rendered output
            buffer += "Found old links on <a href='%s'>%s</a><br>\n" % (obj.absolute_
            ↪url(), obj.Title())
        except:
            pass # Something may fail here if the content object is broken

return buffer
```

More info:

- <http://blog.mfabrik.com/2011/02/17/finding-arbitrary-html-snippets-on-plone-content-pages/>

Other notes

- Indexing tutorial on plone.org
- Manual sorting example

- Getting all unique keywords

Using external catalogs

Description

The Plone catalog can be extend to use external catalogs like Solr or Elasticsearch. Add-ons like `collective.solr` use that to hook into the catalog API and do some Indexing outside of Plone in Solr, which increases performance and flexibility of indexing a lot.

Implement IIndexingQueueProcessor

To hook into the catalog one can implement the `IIndexingQueueProcessor` interface from `Products.CMFCore`.

New in version 5.1: For Plone versions **before 5.1** you need to use the interfaces from `collective.indexing` package!

```
class IIndexQueueProcessor(IIndexing):
    """A queue processor, i.e. an actual implementation of index operations
    for a particular search engine, e.g. the catalog, solr etc
    """

    def begin():
        """Called before processing of the queue is started"""

    def commit():
        """Called after processing of the queue has ended"""

    def abort():
        """Called if processing of the queue needs to be aborted"""
```

Implement IIndexing

And also the underlying `IIndexing` interface.

```
class IIndexing(Interface):
    """ interface for indexing operations, used both for the queue and
    the processors, which perform the actual indexing; the queue gets
    registered as a utility while the processors (portal catalog, solr)
    are registered as named utilities """

    def index(obj, attributes=None):
        """ queue an index operation for the given object and attributes """

    def reindex(obj, attributes=None):
        """ queue a reindex operation for the given object and attributes """

    def unindex(obj):
        """ queue an unindex operation for the given object """
```

Example implementation

For an example implementation of an external `IndexingQueueProcessor`, look at the `SolrIndexProcessor` of `collective.solr`, which implements `ISolrIndexQueueProcessor` (`IIndexQueueProcessor`).

Catalog queue and tests

When running tests, specially during the transition to Plone 5.1, some tests might fail due to the queue holding back and processing until a transaction happens.

If that behavior is not desired one can use the `CATALOG_OPTIMIZATION_DISABLED` environment variable to disable the catalog queue.

```
CATALOG_OPTIMIZATION_DISABLED=y ./bin/test
```

Internationalization (i18n)

There are several layers involved in the processes that provide internationalization capabilities to Plone. Basically they are divided in the ones responsible to translate the user interface and the display of the localization particularities (dates, etc):

- Translating user interface text strings by using `term:gettext`, like the `zope.i18n` and `zope.i18nmessageid` packages.
- Adapting locale-specific settings (such as the time format) for the site, like the `plone.i18n` package.

And the ones responsible for translating the user generated content. Since Plone 5, this is done out of the box with `plone.app.multilingual`

- `plone.app.multilingual` (Archetypes and Dexterity content types, requires at least Plone 4.1)

Contents

Translating text strings

Description

Translating Python and TAL template source code text strings using the `term:gettext` framework and other Plone/Zope `term:i18n` facilities.

Introduction

Internationalization is a process to make your code locale- and language-aware. Usually this means supplying translation files for text strings used in the code.

Plone internally uses the UNIX standard `term:gettext` tool to perform *i18n*.

There are two separate `gettext` systems. Both use the `.po` file format to describe translations.

Note that this chapter concerns only *code-level* translations. *Content* translations are managed by the `plone.app.multilingual` add-on product.

zope.i18n

See also [zope.i18n on pypi](#)

- Follows term:*gettext* best practices
- Translations are stored in the `locales` folder of your application. Example: `locales/fi/LC_MESSAGES/your.app.po`
- Has `zope.i18nmessageid` package, which provides a string-like class which allows storing the translation domain with translatable text strings.
- `.po` files must usually be manually converted to `.mo` binary files every time the translations are updated. See [i18ndude](#). (It is also possible to set an environment variable to trigger recompilation of `.mo` files; see below.)

Plone (at least 3.3) uses only filename and path to search for the translation files. Information in the `.po` file headers is ignored.

Generating a `.pot` template file for your package(s)

`infrae.i18nextextract` can be used in your buildout to create a script which searches particular packages for translation strings. This can be particularly useful for creating a single *translations* package which contains the translations for the set of packages which make up your application.

Add the following to your `buildout.cfg`:

```
[translation]
recipe = infrae.i18nextextract
packages =
    myapplication.policy
    myapplication .theme
output = ${buildout:directory}/src/myapplication.translation/myapplication/
↳translation/locales
output-package = myapplication.translations
domain = mypackage
```

Running the `./bin/translation-extract` script will produce a `.pot` file in the specified output directory which can then be used to create the `.po` files for each translation:

```
msginit --locale=fr --input=locales/mypackage.pot --output=locales/fr/LC_MESSAGES/
↳mypackage.po
```

The `locales` directory should contain a directory for each language, and a directory called `LC_MESSAGES` within each of these, followed by the corresponding `.po` files containing the translation strings:

```
./locales/en/LC_MESSAGES/mypackage.po
./locales/fi/LC_MESSAGES/mypackage.po
./locales/ga/LC_MESSAGES/mypackage.po
```

Marking translatable strings in Python

Each module declares its own `MessageFactory` which is a callable and marks strings with translation domain. `MessageFactory` is declared in the main `__init__.py` file of your package.

```
from zope.i18nmessageid import MessageFactory

# your.app.package must match domain declaration in .po files
MessageFactory = MessageFactory('yourpackage.name')
```

You also need to have the following ZCML entry:

```
<configure xmlns:i18n="http://namespaces.zope.org/i18n">
  <i18n:registerTranslations directory="locales" />
</configure>
```

After the setup above you can use message factory to mark strings with translation domains. `i18ndude` translation utilities use underscore `_` to mark translatable strings (term:*gettext* message ids). Message ids must be unicode strings.

```
from your.app.package import yourAppMessageFactory as _
my_translatable_text = _(u"My text")
```

The object will still look like a string:

```
>>> my_translatable_text
u'My text'
```

But in reality it is a `zope.i18nmessageid.message.Message` object:

```
>>> my_translatable_text.__class__
<type 'zope.i18nmessageid.message.Message'>

>>> my_translatable_text.domain
'your.app.package'
```

To see the translation:

```
>>> from zope.i18n import translate
>>> translate(my_translatable_text)
u"The text of the translation." # This is the corresponding msgstr from the .po file
```

Marking translatable strings in TAL page templates

Declare XML namespace `i18n` and translation domain at the beginning of your template, at the first element

```
<div id="mobile-header" xmlns:i18n="http://xml.zope.org/namespaces/i18n" i18n:domain=
  ↪ "plomobile">
```

Translate element content text using `i18n:translate=""`. It will use the text content of the element as msgid.

```
<li class="heading" i18n:translate="">
  Sections
</li>
```

- Use attributes `i18n:translate`, `i18n:attributes` and so on

For examples look at any core Plone .pt files

Automatically translated message ids

Plone will automatically perform translation for message ids which are output in page templates.

The following code would translate `my_translateable_text` to the native language activated for the current page.

```
<span tal:content="view/my_translateable_text">
```

Note: Since `my_translateable_text` is a `zope.i18nmessageid.message.Message` instance containing its own `gettext` domain information, the `i18n:domain` attribute in page templates does not affect message ids declared through message factories.

Manually translated message ids

If you need to manipulate translated text outside page templates, you need to perform the final translation manually.

Translation always needs context (i.e. under which site the translation happens), as the active language and other preferences are read from the HTTP request object and site object settings.

Translation can be performed using the `context.translate()` method:

```
# Translate some text
msgid = _(u"My text") # my_text is zope.

# Use inherited translate() function to get the final text string
translated = self.context.translate(msgid)

# translated is now u"Käännetty teksti" (in Finnish)
```

`context.translate()` uses the `translate.py` Python script from `LanguageTool`.

It has the signature:

```
def translate(self, domain, msgid, mapping=None, context=None,
              target_language=None, default=None):
```

and does the trick:

```
from Products.CMFCore.utils import getToolByName

# get tool
tool = getToolByName(context, 'translation_service')

# this returns type unicode
value = tool.translate(msgid,
                       domain,
                       mapping,
                       context=context,
                       target_language=target_language,
                       default=default)
```

Note: Translation needs HTTP request object and thus may not work correctly from command-line scripts.

Non-python message ids

There are also other message id markers in code outside the Python domain, that have their own mechanisms:

- ZCML entries
- GenericSetup XML
- TAL page templates

Translating browser view names

Often you might want to translate browser view names, so that the “Display” contentmenu shows something more human readable than, for example, “my_awesome_view”.

These are the steps needed to get it translated:

- Use the “plone” domain for your browser view name translations. Wether put the whole ZCML in the plone domain of just the view definitions with `il8n:domain="plone"`.
- The msgids for the views are their names. Translate them in a plone.po override file in your locales folder.

Please note, `il8ndude` does not parse the `zcml` files for translation strings (see below “Translating other ZCML”).

Translating other ZCML

<http://stackoverflow.com/questions/6899708/do-zcml-files-get-parsed-il8n-wise>

Testing translations

Here is a simple way to check if your gettext domains are correctly loaded.

Plone 4

You can start the Plone debug shell and manually check if translations can be performed.

First start Plone in debug shell:

```
bin/instance debug
```

and then call translation service, in your site, manually:

```
>>> site = app.yoursiteid
>>> translation_service = site.translation_service
>>> translation_service.translate("Add Events Portlet", domain="plone", target_
↪ language="fi")
u'Lis\xe4\xe4 Tapahtumasovelma'
```

Translation string substitution

Translation string substitutions must be used when the final translated message contains *variable strings*.

Plone content classes inherit the `translate()` function which can be used to get the final translated string. It will use the currently activate language. Translation domain will be taken from the `msgid` object itself, which is a string-like `zope.i18nmessageid` instance.

Message ids are immutable (read-only) objects so you need to always create a new message id if you use different variable substitution mappings.

Python code:

```
from saariselka.app import appMessageFactory as _

class SomeView(BrowserView):

    def do_stuff(self):

        msgid = _(u"search_results_found_msg", default=u"Found ${results} results",
↳mapping={u"results": len(self.contents)})

        # Use inherited translate() function to get the final text string
        translated = self.context.translate(msgid)

        # Show the final result count to the user as a portal status message
        messages = IStatusMessage(self.request)
        messages.addStatusMessage(translated, type="info")
```

Corresponding `.po` file entry:

```
#. Default: "Found ${results} results"
#: ./browser/accommodationsummaryview.py:429
msgid "search_results_found_msg"
msgstr "Löytyi ${results} majoituskohdetta"
```

For more information, see

- <http://wiki.zope.org/zope3/TurningMessageIDsIntoRocks>

i18ndude

i18ndude is a developer-oriented command-line utility to manage `.po` and `.mo` files.

Usually you build our own shell script wrapper around *i18ndude* to automate generation of `.mo` files of your product `.po` files.

Note: Plone 3.3 and onwards do not need manual `.po -> .mo` compilation. It is done on start up. Plone 4 has a special switch for this: in your `buildout.cfg` in the part using `plone.recipe.zope2instance` you can set an environment variable for this:

```
environment-vars =
    zope_i18n_compile_mo_files true
```

Note that the value does not matter: the code in `zope.i18n` simply looks for the existence of the variable and does not care what its value is.

Note: If you use *i18ndude* make sure to use `_` as an alias for your `MessageFactory` else *i18ndude* won't find your

message strings in python code and report that “no entries for domain” were found.

See:

- <http://vincentfretin.ecreall.com/articles/my-translation-doesnt-show-up-in-plone-4>

Examples:

- [i18ndude Python package](#)
- [i18ndude example for Plone 3.0 and later](#)
- [i18ndude example for Plone 2.5](#)

Installing i18ndude

The recommended method is to have term:*i18ndude* installed via your [buildout](#).

Add the following to your buildout.cfg:

```
parts =
    ...
    i18ndude

[i18ndude]
unzip = true
recipe = zc.recipe.egg
eggs = i18ndude
```

After this *i18ndude* is available in your buildout/bin folder

```
bin/i18ndude -h
Usage: i18ndude command [options] [path | file1 file2 ...]
```

You can also call it relative to your current package source folder

```
server:home moo$ cd src/mfabrik.plonezohintegration/
server:mfabrik.plonezohintegration moo$ ../../bin/i18ndude
```

Warning: Do not `easy_install i18ndude`. *i18ndude* depends on various Zope packages and pulling them to your system-wide Python configuration could be dangerous, due to potential conflicts with corresponding, but different versions, of the same packages used with Plone.

More information

- <http://markmail.org/message/gru5oaxdl452ekh6#query:+page:1+mid:m22a2ap4xwtwogs5+state:results>

Setting up folder structure for Finnish and English

Example:

```
mkdir locales
mkdir locales/fi
mkdir locales/en
```



```
mkdir locales/fi/LC_MESSAGES
mkdir locales/en/LC_MESSAGES
```

Creating .pot base file

Example:

```
i18ndude rebuild-pot --pot locales/mydomain.pot --create your.app.package .
```

Manual .po entries

i18ndude scans source .py and .pt files for translatable text strings. On some occasions this is not enough - for example if you dynamically generate message ids in your code. Entries which cannot be detected by automatic code scan are called *manual po entries*. They are managed in `locales/manual.pot` which is merged to generated `locales/yournamespace.app.pot` file.

Here is a sample `manual.pot` file:

```
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=utf-8\n"
"Content-Transfer-Encoding: 8bit\n"
"Plural-Forms: nplurals=1; plural=0\n"
"Preferred-Encodings: utf-8 latin1\n"
"Domain: mfabrik.app\n"

# This entry is used in gomobiletheme.mfabrik templates for the campaign page header
# It is not automatically picked, since it is referred from external package
#. Default: "Watch video"
msgid "watch_video"
msgstr ""
```

Managing .po files

Example shell script to manage i18n files. Change CATALOGNAME to reflect the actual package of your product:

The script will:

- pick up all changes to i18n strings in code and reflect them back to the translation catalog of each language;
- pick up changes in `manual.pot` file and reflect them back to the translation catalog of each language.

```
#!/bin/sh
#
# Shell script to manage .po files.
#
# Run this file in the folder main __init__.py of product
#
# E.g. if your product is yourproduct.name
# you run this file in yourproduct.name/yourproduct/name
#
#
# Copyright 2010 mFabrik http://mfabrik.com
```

```
#
# https://plone.org/documentation/manual/plone-community-developer-documentation/i18n/
# ↳ localization
#

# Assume the product name is the current folder name
CURRENT_PATH=`pwd`
CATALOGNAME="yourproduct.app"

# List of languages
LANGUAGES="en fi de"

# Create locales folder structure for languages
install -d locales
for lang in $LANGUAGES; do
    install -d locales/$lang/LC_MESSAGES
done

# Assume i18ndude is installed with buildout
# and this script is run under src/ folder with two nested namespaces in the package_
# ↳ name (like mfabrik.plonezohointegration)
I18NDUDE=../../../../../bin/i18ndude

if test ! -e $I18NDUDE; then
    echo "You must install i18ndude with buildout"
    echo "See https://github.com/collective/collective.developermanual/blob/
    ↳ master/source/i18n/localization.txt"
    exit
fi

#
# Do we need to merge manual PO entries from a file called manual.pot.
# this option is later passed to i18ndude
#
if test -e locales/manual.pot; then
    echo "Manual PO entries detected"
    MERGE="--merge locales/manual.pot"
else
    echo "No manual PO entries detected"
    MERGE=""
fi

# Rebuild .pot
$I18NDUDE rebuild-pot --pot locales/$CATALOGNAME.pot $MERGE --create $CATALOGNAME .

# Compile po files
for lang in $(find locales -mindepth 1 -maxdepth 1 -type d); do

    if test -d $lang/LC_MESSAGES; then

        PO=$lang/LC_MESSAGES/${CATALOGNAME}.po

        # Create po file if not exists
        touch $PO

        # Sync po file
        echo "Syncing $PO"
```

```

i18ndude sync --pot locales/${CATALOGNAME}.pot $PO

# Plone 3.3 and onwards do not need manual .po -> .mo compilation,
# but it will happen on start up if you have
# registered the locales directory in ZCML
# For more info see http://vincentfretin.ecreall.com/articles/my-translation-
↳ doesnt-show-up-in-plone-4

# Compile .po to .mo
# MO=$lang/LC_MESSAGES/${CATALOGNAME}.mo
# echo "Compiling $MO"
# msgfmt -o $MO $lang/LC_MESSAGES/${CATALOGNAME}.po
fi
done

```

Note: Remember to register the locales directory in `configure.zcml` for automatic `.mo` compilation as instructed above.

More information

- http://plataforma.cenditel.gob.ve/browser/proyectosInstitucionales/eGov/ppm/trunk/rebuild_i18n
- <http://encolpe.wordpress.com/2008/04/28/manage-your-internationalization-with-i18ndude/>

Distributing compiled translations

The rule for compiled `.mo` files is that

- Source code repositories (SVN, Git) must not contain compiled `.mo` files
- Released eggs on PyPi, however, **must** contain compiled `.mo` files

The easiest way to manage this is to use the `zest.releaser` tool together with `zest.pocompile` package to release your eggs.

Dynamic content

If your HTML template contains dynamic content such as

```

<h1 i18n:translate="search_form_heading">Search from <span tal:content="context/
↳ @@plone_portal_state/portal_title" /></h1>

```

it will produce `.po` entry:

```
msgstr "Hae sivustolta <span>${DYNAMIC_CONTENT}</span>"
```

You need to give the name to the dynamic part

```

<h1 i18n:translate="search_form_heading">
Search from
<span i18n:name="site_title"
tal:content="context/@@plone_portal_state/portal_title" /></h1>

```

... and then you can refer the dynamic part by a name:

```
#. Default: "Search from <span>${site_title}</span>"
#: ./skins/gomobiletheme_basic/search.pt:46
#: ./skins/gomobiletheme_plone3/search.pt:46
msgid "search_form_heading"
msgstr "Hae sivustolta ${site_title}"
```

More info

- <http://permalink.gmane.org/gmane.comp.web.zope.plone.collective.cvs/111531>

Overriding translations

If you need to change a translation from a `.po` file, you could create a new python package and register your own `.po` files.

To do this, create the package and add a `locales` directory in there, along the lines of what `plone.app.locales` does. Then you can add your own translations in the language that you need; for example `locales/fr/LC_MESSAGES/plone.po` to override French messages in the `plone` domain.

Reference the translation in `configure.zcml` of your package:

```
<configure xmlns:i18n="http://namespaces.zope.org/i18n"
            i18n_domain="my.package">
  <i18n:registerTranslations directory="locales" />
</configure>
```

Your ZCML needs to be included *before* the one from `plone.app.locales`: the first translation of a `msgid` wins. To manage this, you can include the ZCML in the buildout:

```
[instance]
recipe = plone.recipe.zope2instance
user = admin:admin
http-address = 8280
eggs =
    Plone
    my.package
    ${buildout:eggs}
environment-vars =
    zope_i18n_compile_mo_files true
# my.package is needed here so its configure.zcml
# is loaded before plone.app.locales
zcml = my.package
```

See the *Overriding Translations* section of Maurits van Rees's [blog entry on Plone i18n](#), and Vincent Fretin's [posting](#) on the Plone-Users mailing list.

Other

- <http://reinout.vanrees.org/weblog/2007/12/14/translating-schemata-names.html>
- <https://plone.org/products/archgenxml/documentation/how-to/handling-i18n-translation-files-with-archgenxml-and-i18ndude/view?searchterm=>
- <http://vincentfretin.ecreall.com/articles/my-translation-doesnt-show-up-in-plone-4>

Language functions

Note: TODO: rework this section.

Description

Accessing and changing the language state of Plone programmatically.

Introduction

Each page view has a language associated with it.

The active language is negotiated by the `plone.i18n.negotiator` module. Several factors may be involved in determining what the language should be:

- Cookies (setting from the language selector)
- The top-level domain name (e.g. `.fi` for Finnish, `.se` for Swedish)
- Context (current content) language
- Browser language headers

Language is negotiated at the beginning of the page view.

Languages are managed by `portal_languagetool`.

Getting the current language

Example view/viewlet method of getting the current language.

```
from Products.Five.browser import BrowserView
from zope.component import getMultiAdapter

class MyView(BrowserView):
    ...

    def language(self):
        """
        @return: Two-letter string, the active language code
        """
        context = self.context.aq_inner
        portal_state = getMultiAdapter((context, self.request), name=u'plone_portal_
↪state')
        current_language = portal_state.language()
        return current_language
```

Getting language of content item

All content objects don't necessarily support the `Language()` look-up defined by the `IDublinCore` interface. Below is the safe way to extract the served language on the content.

Example BrowserView method:

```
from Acquisition import aq_inner

def language(self):
    """ Get the language of the context.

    Useful in producing <html> tag.
    You need to output language for every HTML page, see http://www.w3.org/TR/xhtml1/
    ↪ #strict

    @return: The two letter language code of the current content.
    """
    portal_state = self.context.unrestrictedTraverse("@@plone_portal_state")

    return aq_inner(self.context).Language() or portal_state.default_language()
```

Getting available site languages

Example below:

```
# Python 2.6 compatible ordered dict
# NOTE: API is not 1:1, but for normal dict access of
# set member, iterate keys and values this is enough
try:
    from collections import OrderedDict
except ImportError:
    from odict import odict as OrderedDict

def getLanguages(self):
    """
    Return list of active languages as ordered dictionary, the preferred first_
    ↪ language as the first.

    Example output::

        {
            u'fi': {u'id' : u'fi', u'flag': u'++resource++country-flags/fi.gif', u
    ↪ 'name': u'Finnish', u'native': u'Suomi'},
            u'de': {u'id' : u'de', u'flag': u'++resource++country-flags/de.gif', u
    ↪ 'name': u'German', u'native': u'Deutsch'},
            u'en': {u'id' : u'en', u'flag': u'++resource++country-flags/gb.gif', u
    ↪ 'name': u'English', u'native': u'English'},
            u'ru': {u'id' : u'ru', u'flag': u'++resource++country-flags/ru.gif', u
    ↪ 'name': u'Russian', u'native': u'\u0420\u0443\u0441\u0441\u043a\u0438\u0439'}
        }
    """
    result = OrderedDict()

    portal_languages = self.context.portal_languages

    # Get barebone language listing from portal_languages tool
    langs = portal_languages.getAvailableLanguages()

    preferred = portal_languages.getPreferredLanguage()

    # Preferred first
```

```

for lang, data in langs.items():
    if lang == preferred:
        result[lang] = data

# Then other languages
for lang, data in langs.items():
    if lang != preferred:
        result[lang] = data

# For convenience, include the language ISO code in the export,
# so it is easier to iterate data in the templates
for lang, data in result.items():
    data["id"] = lang

return result

```

Simple language conditions in page templates

You can do this if full translation strings are not worth the trouble:

```

<div class="main-text">
  <a tal:condition="python:context.restrictedTraverse('@@plone_portal_state').
  ↳ language() == 'fi'" href="http://www.saariselka.fi/sisalto?force-web">Siirry_
  ↳ täydelle web-sivustolle</a>
  <a tal:condition="python:context.restrictedTraverse('@@plone_portal_state').
  ↳ language() != 'fi'" href="http://www.saariselka.fi/sisalto?force-web">Go to full_
  ↳ website</a>
</div>

```

Set site language settings

Manually:

```

# Setup site language settings
portal = context.getSite()
ltool = portal.portal_languages
defaultLanguage = 'en'
supportedLanguages = ['en', 'es']
ltool.manage_setLanguageSettings(defaultLanguage, supportedLanguages,
                                setUseCombinedLanguageCodes=False)

```

For unit testing, you need to run this in `afterSetUp()` after setting up the languages:

```

# THIS IS FOR UNIT TESTING ONLY
# Normally called by pretraverse hook,
# but must be called manually for the unit tests
# Goes only for the current request
ltool.setLanguageBindings()

```

Using `GenericSetup` and `proptiestool.xml`

```

<object name="portal_properties" meta_type="Plone Properties Tool">
  <object name="site_properties" meta_type="Plone Property Sheet">
    <property name="default_language" type="string">en</property>
  </object>
</object>

```

```
</object>
</object>
```

Customizing language selector

Making language flags point to different top level domains

If you use multiple domain names for different languages it is often desirable to make the language selector point to a different domain. Search engines do not really like the dynamic language switchers and will index switching links, messing up your site search results.

Example: TODO

Login-aware language negotiation

By default, language negotiation happens before authentication. Therefore, if you wish to use authenticated credentials in the negotiation, you can do the following.

Hook the after-traversal event.

Example event registration

```
<configure
  xmlns="http://namespaces.zope.org/zope"
  xmlns:browser="http://namespaces.zope.org/browser"
  xmlns:zcml="http://namespaces.zope.org/zcml"
>
  <subscriber handler=".language_negotiation.Negotiator"/>
</configure>
```

Corresponding event handler:

```
from zope.interface import Interface
from zope.component import adapter
from ZPublisher.interfaces import IPubEvent, IPubAfterTraversal
from Products.CMFCore.utils import getToolByName
from AccessControl import getSecurityManager
from zope.app.component.hooks import getSite

@adapter(IPubAfterTraversal)
def Negotiator(event):

    # Keep the current request language (negotiated on portal_languages)
    # untouched

    site = getSite()
    ms = getToolByName(site, 'portal_membership')
    member = ms.getAuthenticatedMember()
    if member.getUserName() == 'Anonymous User':
        return

    language = member.language
    if language:
        # Fake new language for all authenticated users
        event.request['LANGUAGE'] = language
```



```
event.request.LANGUAGE_TOOL.LANGUAGE = language
else:
    lt = getToolByName(site, 'portal_languages')
    event.request['LANGUAGE'] = lt.getDefaultLanguage()
    event.request.LANGUAGE_TOOL.LANGUAGE = lt.getDefaultLanguage()
```

Other

- <http://reinout.vanrees.org/weblog/2007/12/14/translating-schemata-names.html>
- <http://maurits.vanrees.org/weblog/archive/2007/09/i18n-locales-and-plone-3.0>
- <http://blogs.ingeniweb.com/blogs/user/7/tag/i18ndude/>
- <https://plone.org/products/archgenxml/documentation/how-to/handling-i18n-translation-files-with-archgenxml-and-i18ndude/view?searchterm=>

Translated content

Description

Translating content items in Plone, creating translations programmatically and working with translators.

Introduction

Plone ships out of the box with a multilingual solution for translating user generated content.

For all practical purposes, you should use that package, `plone.app.multilingual`

Note: For earlier Plone versions, there were other solutions like `LinguaPlone` and `raptus.multilanguageplone`. Refer to the [Plone 4 version of this document](#) if you need that information.

`plone.app.multilingual`

`plone.app.multilingual` was designed originally to provide Plone a whole multilingual story. Using ZCA technologies, enables translations to Dexterity and Archetypes content types as well managed via an unified UI.

This module provides the user interface for managing content translations. It's the app package of the next generation Plone multilingual engine. It's designed to work with Dexterity content types and the *old fashioned* Archetypes based content types as well. It only works with Plone 4.1 and above due to the use of UUIDs for referencing the translations.

For more information see `plone.app.multilingual`

Installation

To use this package with both Dexterity and Archetypes based content types you should add the following line to your *eggs* buildout section:

```
eggs =
    plone.app.multilingual[archetypes, dexterity]
```

If you need to use this package only with Archetypes based content types you only need the following line:

```
eggs =
    plone.app.multilingual[archetypes]
```

While archetypes is default in Plone for now, you can strip `[archetypes]`. This may change in future so we recommend adding an appendix as shown above.

Setup

After re-running your buildout and installing the newly available add-ons, you should go to the *Languages* section of your site's control panel and select at least two or more languages for your site. You will now be able to create translations of Plone's default content types, or to link existing content as translations.

Marking objects as translatable

Archetypes

By default, if PAM is installed, Archetypes-based content types are marked as translatable

Dexterity

Users should mark a dexterity content type as translatable by assigning a the multilingual behavior to the definition of the content type either via file system, supermodel or through the web.

Marking fields as language independent

Archetypes

The language independent fields on Archetype-based content are marked as follows (same as in previous version of Plone with LinguaPlone in place):

```
atapi.StringField(
    'myField',
    widget=atapi.StringWidget(
        ....
    ),
    languageIndependent=True
),
```

Dexterity

There are four ways of achieve it.

Grok directive

In your content type class declaration:

```
from plone.app.multilingual.dx import directives
directives.languageindependent('field')
```

Supermodel

In your content type XML file declaration:

```
<field name="myField" type="zope.schema.TextLine" lingua:independent="true">
  <description />
  <title>myField</title>
</field>
```

Native

In your code:

```
from plone.app.multilingual.dx.interfaces import ILanguageIndependentField
alsoProvides(ISchema['myField'], ILanguageIndependentField)
```

Through the web

Via the content type definition in the *Dexterity Content Types* control panel.

Language get/set via an unified adapter

In order to access and modify the language of a content type regardless the type (Archetypes/Dexterity) there is a interface/adapter:

```
plone.app.multilingual.interfaces.ILanguage
```

You can use:

```
from plone.app.multilingual.interfaces import ILanguage
language = ILanguage(context).get_language()
```

or in case you want to set the language of a content:

```
language = ILanguage(context).set_language('ca')
```

ITranslationManager adapter

The most interesting adapter that p.a.m. provides is: `plone.app.multilingual.interfaces.ITranslationManager`.

It adapts any ITranslatable object to provide convenience methods to manage the translations for that object.

Add a translation

Given an object *obj* and we want to translate it to Catalan language ('ca'):

```
from plone.app.multilingual.interfaces import ITranslationManager
ITranslationManager(obj).add_translation('ca')
```

Register a translation for an already existing content

Given an object *obj* and we want to add *obj2* as a translation for Catalan language ('ca'):

```
ITranslationManager(obj).register_translation('ca', obj2)
```

Get translations for an object

Given an object *obj*:

```
ITranslationManager(obj).get_translations()
```

and if we want a concrete translation:

```
ITranslationManager(obj).get_translation('ca')
```

Check if an object has translations

Given an object *obj*:

```
ITranslationManager(obj).get_translated_languages()
```

or:

```
ITranslationManager(obj).has_translation('ca')
```

For more information see: <https://github.com/plone/plone.app.multilingual/blob/master/src/plone/app/multilingual/interfaces.py#L76>

How to contribute to Plone core translations

Description

How to contribute to the Plone translations.

Introduction

You need to have write access to <https://github.com/collective/plone.app.locales> to be able to commit your translation directly. You can also update a po file online and make a pull request.

Updating translations for Plone 4.2

To do.

Updating translations for Plone 4.3

If you want to test your latest translation with unreleased packages containing i18n fixes for Plone 5, get the buildout like this:

```
git clone -b 4.2 git://github.com/plone/buildout.coredev.git
cd buildout.coredev
python2.7 bootstrap.py
bin/buildout -c experimental/i18n.cfg
rm .mr.developer.cfg
ln -s experimental/.mr.developer.cfg
bin/instance fg
```

To update the buildout later:

```
git pull
bin/develop up -f
```

To update your translation, you can go there:

```
cd src/plone.app.locales/plone/app/locales/
```

Here you have the following directories:

- `locales` used for core Plone translations.
- `locales-addons` used for some addons packages.
- `locales-future` used for Plone 5 packages. The po files can change really often. The translations will normally be merged to the `locales` directory when Plone 5 will be released. This directory is not included in `plone.app.locales` 4.x releases. For developers: if you want to add a Plone 5 package to this directory, you can add it to the `plone5` variable in `experimental/i18n.cfg`, don't forget to add a line in `[sources]` if it's missing.

Open the po file with poedit, kbabel or any other i18n tool. For example for French:

```
poedit locales/fr/LC_MESSAGES/plone.po
```

Please do a `git pull` before editing a po file to be sure you have the latest version.

Committing directly (commit access)

You can commit your translation from this locales directory:

```
git commit -a -m "Updated French translation"
git push
```

Creating a pull request (no commit access)

If you do not have commit access on GitHub [collective group](#), you can do the following:

Login to GitHub. Go to GitHub [plone.app.locales](#)

Press *Fork*. Now GitHub creates a copy of `plone.app.locales` package for you.

Then on your computer in `plone.app.locales` do a special git push to your own repository:

```
git push git@github.com:YOURUSERNAMEHERE/plone.app.locales.git
```

Go to GitHub <https://github.com/YOURUSERNAME/plone.app.locales>

Press button *Create Pull request*. Fill it in.

The request will appear for *plone.app.locales* authors. If it does not get merged in timely manner, poke people on the #plone IRC channel or the mailing list below (sometimes requests go unnoticed).

Support

Please ask questions on the [plone-i18n mailing-list](#).

Users and members

Member manipulation

Description

How to programmatically create, read, edit and delete site members.

Introduction

In Plone, there are two loosely-coupled subsystems relating to members:

Authentication and permission information (`acl_users` under site root), managed by the *PAS*. In a default installation, this corresponds to Zope user objects. PAS is *pluggable*, though, so it may also be authenticating against an LDAP server, Plone content objects, or other sources.

Member profile information accessible through the `portal_membership` tool. These represent Plone members. PAS authenticates, and the Plone member object provides metadata about the member.

Getting the logged-in member

Anonymous and logged-in members are exposed via the *IPortalState context helper*.

Example (browserview: use `self.context` since `self` is not acquisition-wrapped):

```
from zope.component import getMultiAdapter

portal_state = getMultiAdapter(
    (self.context, self.request), name="plone_portal_state")
if portal_state.anonymous():
    # Return target URL for the site anonymous visitors
    return self.product.getHomepageLink()
else:
```

```
# Return edit URL for the site members
return product.absolute_url()
```

or from a template:

```
<div tal:define="username context/portal_membership/getAuthenticatedMember/getUserName
↪">
    ...
</div>
```

Getting any member

To get a member by username (you must have Manager role):

```
mt = getToolByName(self.context, 'portal_membership')
member = mt.getMemberById(username)
```

To get all usernames:

```
mt = getToolByName(self.context, 'portal_membership')
memberIds = mt.listMemberIds()
```

Getting member information

Once you have access to the member object, you can grab basic information about it.

Get the user's name:

```
member.getName()
```

Resetting user password without emailing them

- <https://plone.org/documentation/kb/reset-a-password-without-having-to-email-one-to-the-user>

Exporting and importing member passwords

You can also get at the hash of the user's password (only the hash is available, and only for standard Plone user objects) (in this example we're in Plone add-on context, since `self` is acquisition-wrapped):

```
uf = getToolByName(self, 'acl_users')
passwordhash_map = uf.source_users._user_passwords
userpasswordhash = passwordhash_map.get(member.id, '')
```

Note that this is a private data structure. Depending on the Plone version and add-ons in use, it may not be available.

You can use this hash directly when importing your user data, for example as follows (can be executed from a *debug prompt*):

```
# The file 'exported.txt' contains lines with: "memberid hash"
lines = open('exported.txt').readlines()
changes = []
```

```
c = 0
members = mt.listMembers()
for l in lines:
    memberid, passwordhash_exported = l.split(' ')
    passwordhash_exported = passwordhash_exported.strip()
    member = mt.getMemberById(memberid)
    if not member:
        print 'missing', memberid
        continue
    passwordhash = passwordhash_map.get(memberid)
    if passwordhash != passwordhash_exported:
        print 'changed', memberid, passwordhash, passwordhash_exported
        c += 1
        changes.append((memberid, passwordhash_exported))

uf.source_users._user_passwords.update(changes)
```

Also, take a look at a script for exporting Plone 3.0's memberdata and passwords:

- <http://blog.kagesenshi.org/2008/05/exporting-plone30-memberdata-and.html>

Iterating all site users

Example:

```
buffer = ""

# Returns list of site usernames
mt = getToolByName(self, 'portal_membership')
users = mt.listMemberIds()
# alternative: get member objects
# members = mt.listMembers()

for user in users:
    print "Got username:" + user
```

Note: Zope users, such as *admin*, are not included in this list.

Getting all *Members* for a given *Role*

In this example we use the `portal_membership` tool. We assume that a role called `Agent` exists and that we already have the context:

```
from Products.CMFCore.utils import getToolByName

membership_tool = getToolByName(self, 'portal_membership')
agents = [member for member in membership_tool.listMembers()
          if member.has_role('Agent')]
```


Groups

Groups are stored as `PloneGroup` objects. `PloneGroup` is a subclass of `PloneUser`. Groups are managed by the `portal_groups` tool.

- <https://github.com/plone/Products.PlonePAS/blob/master/Products/PlonePAS/plugins/ufactory.py>
- <https://github.com/plone/Products.PlonePAS/blob/master/Products/PlonePAS/plugins/group.py>

Creating a group

Example:

```
groups_tool = getToolByName(context, 'portal_groups')

group_id = "companies"
if not group_id in groups_tool.getGroupIds():
    groups_tool.addGroup(group_id)
```

For more information, see:

- https://github.com/plone/Products.PlonePAS/blob/master/Products/PlonePAS/tests/test_groupstool.py
- <https://github.com/plone/Products.PlonePAS/blob/master/Products/PlonePAS/plugins/group.py>

Add local roles to a group

Example:

```
from AccessControl.interfaces import IRoleManager
if IRoleManager.providedBy(context):
    context.manage_addLocalRoles(groupid, ['Manager',])
```

Note: This is an example of code in a *view*, where `context` is available.

Update properties for a group

The `editGroup` method modifies the title and description in the `source_groups` plugin, and subsequently calls `setGroupProperties(kw)` which sets the properties on the `mutable_properties` plugin.

Example:

```
portal_groups.editGroup(groupid, **properties)
portal_groups.editGroup(groupid, roles = ['Manager',])
portal_groups.editGroup(groupid, title = u'my group title')
```

Getting available groups

Getting all groups on the site is possible through `acl_users` and the `source_groups` plugin, which provides the functionality to manipulate Plone groups.

Example to get only ids:

```
acl_users = getToolByName(self, 'acl_users')
# Iterable returning id strings:
groups = acl_users.source_groups.getGroupIds()
```

Example to get full group information:

```
acl_users = getToolByName(self, 'acl_users')
group_list = acl_users.source_groups.getGroups()

for group in group_list:
    # group is PloneGroup object
    yield (group.getName(), group.title)
```

List users within all groups

Example to get the email addresses of all users on a site, by group:

```
acl_users = getToolByName(context, 'acl_users')
groups_tool = getToolByName(context, 'portal_groups')
groups = acl_users.source_groups.getGroupIds()
for group_id in groups:
    group = groups_tool.getGroupById(group_id)
    if group is None:
        continue
    members = group.getGroupMembers()
    member_emails = [m.getProperty('email') for m in members]
    ...
```

Adding a user to a group

Example:

```
# Add user to group "companies"
portal_groups = getToolByName(self, 'portal_groups')
portal_groups.addPrincipalToGroup(member.getUserName(), "companies")
```

Removing a user from a group

Example:

```
portal_groups.removePrincipalFromGroup(member.getUserName(), "companies")
```

Getting groups for a certain user

Below is an example of getting groups for the logged-in user (Plone 3 and earlier):

```
mt = getToolByName(self.context, 'portal_membership')
mt.getAuthenticatedMember().getGroups()
```

In Plone 4 you have to use:

```
groups_tool = getToolByName(self, 'portal_groups')
groups_tool.getGroupsByUserId('admin')
```

Checking whether a user exists

Example:

```
mt = getToolByName(self, 'portal_membership')
return mt.getMemberById(id) is None
```

See also:

- <http://svn.zope.org/Products.CMFCore/trunk/Products/CMFCore/RegistrationTool.py?rev=110418&view=auto>

Creating users

Use the portal_registration tool. Example (browser view):

```
def createCompany(request, site, username, title, email, passwd=None):
    """
    Utility function which performs the actual creation, role and permission magic.

    @param username: Unicode string

    @param title: Fullname of user, unicode string

    @return: Created company content item or None if the creation fails
    """

    # If we use custom member properties they must be initialized
    # before regtool is called
    prepareMemberProperties(site)

    # portal_registration manages new user creation
    regtool = getToolByName(site, 'portal_registration')

    # Default password to the username
    # ... don't do this on the production server!
    if passwd == None:
        passwd = username

    # We allow only lowercase
    username = username.lower()

    # Username must be ASCII string
    # or Plone will choke when the user tries to log in
    try:
        username = str(username)
    except UnicodeEncodeError:
        IStatusMessage(request).addStatusMessage(_(u"Username must contain only_
↳ characters a-z"), "error")
        return None

    # This is the minimum required information
```

```
# to create a working member
properties = {
    'username': username,
    # Full name must always be utf-8 encoded
    'fullname': title.encode("utf-8"),
    'email': email
}

try:
    # addMember() returns MemberData object
    member = regtool.addMember(username, passwd, properties=properties)
except ValueError, e:
    # Give user visual feedback what went wrong
    IStatusMessage(request).addStatusMessage(_(u"Could not create the user:") +
↪unicode(e), "error")
    return None
```

Batch member creation

- An example script can be run with bin/plonectl, tested on Plone 4.3.3; see <http://gist.github.com/l34marr/02a9ef12a1e51c474bee>
- An example script tested on Plone 2.5.x; see <https://plone.org/documentation/kb/batch-adding-users>

Email login

- In Plone 4 and up, it is a default feature.

Custom member creation form: complex example

Below is an example of a Grok form which the administrator can use to create new users. New users will receive special properties and a folder for which they have ownership access. The password is set to be the same as the username. The user is added to a group named “companies”.

Example company.py:

```
""" Add companies.

Create user account + associated "home folder" content type
for a company user.
User accounts have a special role.

Note: As of this writing, in 2010-04, we need the
plone.app.directives trunk version which
contains an unreleased validation decorator.
"""

# Core Zope 2 + Zope 3 + Plone
from zope.interface import Interface
from zope import schema
from five import grok
from Products.CMFCore.interfaces import ISiteRoot
from Products.CMFCore.utils import getToolByName
from Products.CMFCore import permissions
```

```

from Products.statusmessages.interfaces import IStatusMessage

# Form and validation
from z3c.form import field
import z3c.form.button
from plone.directives import form
from collective.z3cform.grok.grok import PloneFormWrapper
import plone.autoform.form

# Products.validation uses some ugly ZService magic which I can't quite comprehend
from Products.validation import validation

# Our translation catalog
from zope.i18nmessageid import MessageFactory
OurMessageFactory = MessageFactory('OurProduct')
OurMessageFactory = _

# If we're building an addon, we may already have one, for example:
# from isleofback.app import appMessageFactory as _

grok.templatedir("templates")

class ICompanyCreationFormSchema(form.Schema):
    """ Define fields used on the form """

    username = schema.TextLine(title=u"Username")

    company_name = schema.TextLine(title=u"Company name")

    email = schema.TextLine(title=u"Email")

class CompanyCreationForm(plone.autoform.form.AutoExtensibleForm, form.Form):
    """ Form action controller.

    form.DisplayForm will automatically expose the form
    as a view, no wrapping view creation needed.
    """

    # Form label
    name = _(u"Create Company")

    # Which schema is used by AutoExtensibleForm
    schema = ICompanyCreationFormSchema

    # The form does not care about the context object
    # and should not try to extract field value
    # defaults out of it
    ignoreContext = True

    # This form is available at the site root only
    grok.context(ISiteRoot)

    # z3c.form has a function decorator
    # which turns the function to a form button action handler

    @z3c.form.button.buttonAndHandler(_('Create Company'), name='create')
    def createCompanyAction(self, action):

```

```
    """ Button action handler to create company.
    """

    data, errors = self.extractData()
    if errors:
        self.status = self.formErrorsMessage
        return

    obj = createCompany(self.request, self.context, data["username"], data[
↪ "company_name"], data["email"])
    if obj is not None:
        # mark as finished only if we get the new object
        IStatusMessage(self.request).addStatusMessage(_(u"Company created"), "info
↪ ")

class CompanyCreationView(PloneFormWrapper):
    """ View which exposes form as URL """

    form = CompanyCreationForm

    # Set up security barrier -
    # non-privileged users can't access this form
    grok.require("cmf.ManagePortal")

    # Use http://yourhost/@@create_company URL to access this form
    grok.name("create_company")

    # This view is available at the site root only
    grok.context(ISiteRoot)

    # Which template is used to decorate the form
    # -> forms.pt in template directory
    grok.template("form")

@form.validator(field=ICompanyCreationFormSchema['email'])
def validateEmail(value):
    """ Use old Products.validation validators to perform the validation.
    """
    validator_function = validation.validatorFor('isEmail')
    if not validator_function(value):
        raise schema.ValidationError(u"Entered email address is not good:" + value)

def prepareMemberProperties(site):
    """ Adjust site for custom member properties """

    # Need to use ancient Z2 property sheet API here...
    portal_memberdata = getToolByName(site, "portal_memberdata")

    # When new member is created, its MemberData
    # is populated with the values from portal_memberdata property sheet,
    # so value="" will be the default value for users' home_folder_uid
    # member property
    if not portal_memberdata.hasProperty("home_folder_uid"):
        portal_memberdata.manage_addProperty(id="home_folder_uid", value="", type=
↪ "string")
```

```

# Create a group "companies" where newly created members will be added
acl_users = getToolByName(site, 'acl_users')
gt = getToolByName(site, 'portal_groups')

group_id = "companies"
if not group_id in gt.getGroupIds():
    gt.addGroup(group_id, [], [], {'title': 'Companies'})

def createCompany(request, site, username, title, email, passwd=None):
    """
    Utility function which performs the actual creation, role and permission magic.

    @param username: Unicode string

    @param title: Fullname of user, unicode string

    @return: Created company content item or None if the creation fails
    """

    # If we use custom member properties
    # they must be initialized before regtool is called
    prepareMemberProperties(site)

    # portal_registrations manages new user creation
    regtool = getToolByName(site, 'portal_registration')

    # Default password to the username
    # ... don't do this on the production server!
    if passwd == None:
        passwd = username

    # Only lowercase allowed
    username = username.lower()

    # Username must be ASCII string
    # or Plone will choke when the user tries to log in
    try:
        username = str(username)
    except UnicodeEncodeError:
        IStatusMessage(request).addStatusMessage(_(u"Username must contain only_
↳characters a-z"), "error")
        return None

    # This is minimum required information set
    # to create a working member
    properties = {
        'username': username,
        # Full name must be always as utf-8 encoded
        'fullname': title.encode("utf-8"),
        'email': email
    }

    try:
        # addMember() returns MemberData object
        member = regtool.addMember(username, passwd, properties=properties)
    except ValueError, e:

```

```
        # Give user visual feedback what went wrong
        IStatusMessage(request).addStatusMessage(_(u"Could not create the user:") +
↳ unicode(e), "error")
        return None

    # Add user to group "companies"
    gt = getToolByName(site, 'portal_groups')
    gt.addPrincipalToGroup(member.getUserName(), "companies")

    return createMatchingHomeFolder(request, site, member)

def createMatchingHomeFolder(request, site, member, target_folder="yritykset", target_
↳ type="IsleofbackCompany", language="fi"):
    """ Creates a folder, sets its ownership for the member and stores the folder UID,
↳ in the member data.

    @param member: MemberData object

    @param target_folder: Under which folder a new content item is created

    @param language: Initial two language code of the item
    """

    parent_folder = site.restrictedTraverse(target_folder)

    # Cannot add custom memberdata properties unless explicitly declared

    id = member.getUserName()

    parent_folder.invokeFactory(target_type, id)

    home_folder = parent_folder[id]
    name = member.getProperty("fullname")

    home_folder.setTitle(name)
    home_folder.setLanguage(language)

    email = member.getProperty("email")
    home_folder.setEmail(email)

    # Unset the Archetypes object creation flag
    home_folder.processForm()

    # Store UID of the created folder in memberdata so we can
    # look it up later to e.g. generate the link to the member folder
    member.setMemberProperties({"home_folder_uid": home_folder.UID()})

    # Get the user handle from member data object
    user = member.getUser()
    username = user.getUserName()

    home_folder.manage_setLocalRoles(username, ["Owner",])
    home_folder.reindexObjectSecurity()

    return home_folder
```


Member profiles

Description

How to manage Plone member properties programmatically

Introduction

Member profile fields are the fields which the logged-in member can edit himself on his user account page.

For more info, see:

MemberDataTool <https://github.com/zopefoundation/Products.CMFCore/blob/master/Products/CMFCore/MemberDataTool.py>

MemberData class <https://github.com/zopefoundation/Products.CMFCore/blob/master/Products/CMFCore/MemberDataTool.py>

PlonePAS subclasses and extends MemberData and MemberDataTool.

- See PlonePAS MemberDataTool.
- See PlonePAS MemberData class.

Getting member profile properties

Note: The following applies to vanilla Plone. If you have customized membership behavior it won't necessarily work.

Member profile properties (title, address, biography, etc.) are stored in `portal_membership` tool.

Available fields can be found in the Management Interface -> `portal_membership` -> *Properties* tab.

The script below is a simple example showing how to list all member email addresses:

```
from Products.CMFCore.utils import getToolByName
memberinfo = []
membership = getToolByName(self.context, 'portal_membership')
for member in membership.listMembers():
    memberinfo.append(member.getProperty('email'))
return memberinfo
```

Accessing member data

Further reading

- **ToolbarViewlet has some sample code** how to retrieve these properties.

Getting member fullname

In Python code you can access properties on the `MemberData` object:

```
fullname = member_data.getProperty("fullname")
```

In a template you can do something along the same lines:

```
<tal:with-fullname define="membership context/portal_membership;info_
↳python:membership.getMemberInfo(user.getId()); fullname info/fullname">
    You are are <span class="name" tal:content="fullname" />
</tal:with-fullname>
```

Note that this code won't work for anonymous users.

Setting member profile properties

Use `setMemberProperties(mapping={...})` to batch update properties. Old properties are not removed.

Example:

```
member = portal_membership.getMemberById(user_id)
member.setMemberProperties(mapping={"email": "aaa@aaa.com"})
```

New properties must be explicitly declared in `portal_memberdata`, before creation of the member, or `setMemberProperties()` will silently fail.

Example:

```
def prepareMemberProperties(site):
    """ Adjust site for custom member properties """

    # Need to use ancient Z2 property sheet API here...
    portal_memberdata = getToolByName(site, "portal_memberdata")

    # When new member is created, it's MemberData
    # is populated with the values from portal_memberdata property sheet,
    # so value="" will be the default value for users' home_folder_uid
    # member property
    if not portal_memberdata.hasProperty("home_folder_uid"):
        portal_memberdata.manage_addProperty(id="home_folder_uid", value="", type=
↳"string")

    ....

def createMatchingHomeFolder(member):
    """ """

    email = member.getProperty("email")
    home_folder.setEmail(email)

    # Store UID of the created folder in memberdata so we can
    # look it up later to e.g. generate the link to the member folder
    member.setMemberProperties(mapping={"home_folder_uid": home_folder.UID()})

    return home_folder
```

Setting password

Password is a special case.

Example how to set the user password:

```
# Password is set in a special way
# passwd is password as plain text
member.setSecurityProfile(password=passwd)
```

Increase minimum password size

To increase the minimum password size, copy `validate_pwreset_password` to your custom folder and insert the following lines:

```
if len(password) < 8:
    state.setError('password', 'ERROR')
```

This will increase the minimum password size for the password reset form to 8 characters. (This does not effect new user registration, that limit will still be 5.)

Don't forget to update your form templates to reflect your changes!

Default password length - password reset form

The password reset form's minimum password length is 5 characters. To increase this:

Copy `validate_pwreset_password` into your custom folder and add the following lines:

```
if len(password) < 8:
    state.setError('password', 'ERROR')
```

before the `if state.getErrors():` method.

This would increase the minimum password size to 8 characters. Remember to update your form templates accordingly.

Setting visual editor for all users

The *visual editor* property is set on the member upon creation.

If you want to change all site members to use TinyMCE instead of Kupu, you have to do it using the command-line — Plone provides no through-the-web way to change the properties of other members. Here is a script which does the job:

`migrate.py`:

```
import transaction

# Traverse to your Plone site from Zope application root
context = app.yoursiteid.sitsngta # site id is yoursiteid

usernames = context.acl_users.getUserNames()
portal_membership = context.portal_membership
txn = transaction.get()
```

```
i = 0
for userid in usernames:
    member = portal_membership.getMemberById(userid)
    value = member.wysiwyg_editor

    # Show the existing editor choice before upgrading
    print str(userid) + ": " + str(value)

    # Set WYSIWYG editor for the member
    member.wysiwyg_editor = "TinyMCE"

    # Make sure transaction buffer does not grow too large
    i += 1
    if i % 25 == 0:
        txn.savepoint(optimistic=True)

# Once done, commit all the changes
txn.commit()
```

Run it:

```
bin/instance run migrate.py
```

Note: The script does not work through the Management Interface as member properties do not have proper security declarations, meaning, no changes are permitted.

Password reset requests

Directly manipulating password reset requests is useful e.g. for testing.

Poking requests:

```
# Poke password reset information
reset_requests = self.portal.portal_password_reset._requests.values()
self.assertEqual(1, len(reset_requests))
# reset requests are keyed by their random magic string
key = self.portal.portal_password_reset._requests.keys()[0]
reset_link = self.portal.pwreset_constructURL(key)
```

Clearing all requests:

```
# Reset all password reset requests
self.portal.portal_password_reset._requests = {}
```

Members as content

Description

The `Products.membrane` and `Products.remember` add-ons provide member management where members are represented by Plone content items. The member-as-content paradigm makes member management radically flexible: members can be in different folders, have different workflows and states and different profile fields.

It is also possible to use this approach with dexterity; for that, use the `dexterity.membrane` add-on.

Introduction

remember (small r) and *membrane* are framework add-on products for Plone which allows you to manipulate site members as normal content objects. The product also allows distributed user management and different user classes.

- `Products.membrane` provides a framework for integrating `acl_users`, which manages access rights, with content-like members and tasks like login.
- `Products.remember` is a basic implementation of this with two different user workflows and a normal user schema.
- `dexterity.membrane` is a port of `Products.membrane` to the dexterity framework.

Basics

- Read the [membrane tutorial](#).
- See the example code `Products.membrane.example`.
- Read the documents at `Products.remember/docs/tutorial`.
- See the [Weblion FacultyStaffDirectory product](#), which is a sophisticated implementation of the framework.
- It is recommended to enable debug-level logging output for membrane related unit tests, as PlonePAS code swallows several exceptions and does not output them unless debug level is activated.

Getting member by username

Example:

```
from Products.CMFCore.utils import getToolByName

membrane = getToolByName(context, "membrane_tool")

# getUserAuthProvider returns None if there is no membrane-based user
# match for username
# e.g. this will return None for Zope admin user
sits_user = membrane.getUserAuthProvider(username)
return sits_user
```

Getting Plone member from MembraneUser or owner record

Below is an example of how to resolve member content object from `MembraneUser` record “owner” who is user “local_user”:

```
(Pdb) mbtool = self.portal.membrane_tool
(Pdb) owner
<MembraneUser 'local_user'>
(Pdb) mbtool.getUserAuthProvider(owner.getId())
<SitsLocalUser at /plone/country/hospital/local_users/local_user
```

Creating a member

The following snippet works in unit tests:

```
mem_password = 'secret'

def_mem_data = {
    'email': 'noreply@xxxxxxxxxyyyyyy.com',
    'password': mem_password,
    'confirm_password': mem_password,
}

mem_data = {
    'portal_member':
        {
            'fullname': 'Portal Member',
            'mail_me': True,
        },
    'admin_member':
        {
            'roles': ['Manager', 'Member']
        },
    'blank_member':
        {},
}

mdata = getToolByName(self.portal, 'portal_memberdata')

mdata.invokeFactory("MyUserPortalType", name)
member = getattr(mdata, name) #
```

Populating member fields automatically

Use the following unit test snippet:

```
def populateUser(self, member):
    """ Auto-populate member object required fields based on Archetypes schema.

    @param member: Membrane member content object
    """

    from Products.SitsHospital.content.SitsUser import SitsUser

    schema = SitsUser.schema

    data = {}

    for f in schema._fields.values():

        if not f.required:
            continue

        if f.__name__ in [ "password", "id" ]:
            # Do not set password or member id
            continue
```

```

# Autofill member field values
if f.vocabulary:
    value = f.vocabulary[0][0]
elif f.__name__ in [ "email" ]:
    value = "test@xyz.com"
else:
    value = "foo"

# print "filling in field:" + str(f)

data[f.__name__] = value

member.update(**data)

```

Checking member validity

The following snippet is useful for unit testing:

```

def assertValidMember(self, member):
    """ Emulate Products.remember.content.member validation behavior with verbose_
    ↪ output.

    """
    errors = {}
    # make sure object has required data and metadata
    member.Schema().validate(member, None, errors, 1, 1)
    if errors:
        raise AssertionError("Member contained errors:" + str(errors))

```

Setting user password

Passwords are stored hashed and can be set using the `BaseMember._setPassword()` method.

`_setPassword()` takes the password as a plain-text argument and hashes it before storing:

```
user_object._setPassword("secret")
```

You may also use the `portal_registrations` tool. This method is security-checked and may be used from Management Interface scripts:

```

rtool = context.portal_registration
rtool.editMember(id, properties={}, password="secret")

```

Use `getToolByName` rather than acquiring the tool from `context` if you're doing this in a browser view.

Accessing hashed password

Use the password attribute directly:

```
hashed = user_object.password
```

The password hash should be a unicode string.

Note: By default, `Products.remember` uses the `HMACHash` hasher. As a salt, the `str(context)` string is used. This means that it is not possible to move hashed password from one context item to another. For more information, see the `Products.remember.content.password_hashers` module.

Moving members

Moving members is not straightforward, as by default member password is hashed with the member location.

- Members need to reregister their password after being moved from one folder to another.

Here is a complex function to perform moving by recreating the user and deleting the old object:

```
import logging

from Products.CMFCore.utils import getToolByName
from Products.Archetypes import public as atapi

from Products.SitsHospital.interfaces import ISitsUser, ISitsLocalUser,
↳ISitsLocalCoordinatorUser

logger = logging.getLogger("RememberUserCopy")

def createUser(sourceUser, username, targetFolder):
    """ Default example user creator """
    targetFolder.invokeFactory("Member", username)
    return targetFolder[username]

def postProcess(sourceUser, targetUser):
    """ Hook to set-up additional fields which do not have 1:1 mapping in the new and
↳old user objects """
    pass

def copyRememberUser(sourceUser, targetFolder, user_constructor=createUser, post_
↳process=postProcess, expected_creation_state="new_private", expected_initialization_
↳state="private"):
    """
    Copies Product.remember based user from one location to another.

    This is useful if you have locally stored members on your site
    (for example one folder per country)
    and you need to move the person from one country to another.

    Member password is hashed against the member object location.
    Thus, the password will be invalid if the physical path of the member object
↳changes.
    All moved members are asked to re-enter their passwords.

    If betahaus.emaillogin is installed we also update its catalog so that
    the email login works after the member has been moved.

    When all the fields in the user schema validate successfully,
    the re-registration email for the new user is automatically send
    (TODO: Not sure whether this is general condition for Products.Remember)
```



```

    @param sourceUser: from Products.remember.content.member.Member instance

    @param targetFolder: Any folderish object which can contain Member instances

    @param user_constructor: function(sourceUser, targetFolder) if special user_
    ↪creation is needed

    @param post_process: function(sourceUser, targetUser) for setting up custom_
    ↪fields if there is no 1:1 mapping between fields of the new and old user object._
    ↪Also you can do workflow mangling here.

    @param expected_creation_state: The workflow state where the new member should be_
    ↪after it has been correctly initialized. In this point update() is not yet called,_
    ↪so Remember automatic registration mechanism should have not been triggered.

    @param expected_initialization_state: The workflow state where the new member_
    ↪should be after it has been correctly initialized. In this point update() is not_
    ↪yet called, so Remember automatic registration mechanism should have not been_
    ↪triggered.

    @return: The newly created national coordinator object.
    """

    # shortcut to the source user
    lc = sourceUser

    # Validate LC user
    errors = {}
    lc.Schema().validate(lc, None, errors, True, True)
    if errors:
        assert not errors, "The source user must be valid before moving. Errors:" +
    ↪str(errors)

    username = lc.getUserName()

    logger.debug("Copying user:" + username)

    # Make sure that LC username is free
    id = lc.getId()
    parent = lc.aq_parent

    assert lc.cb_userHasCopyOrMovePermission(), "No permission"
    assert lc.cb_isMoveable(), "Object problem"

    # We temporarily rename the old object for the duration
    # of the moving so that the id of the member
    # object won't conflict with the newly created target user
    new_id = id + "-old"
    assert type(new_id) != unicode

    parent.manage_renameObject(id, new_id)

    # We need to re-fetch the object handle as it has changed in rename
    lc = parent[new_id]

    # nc = newly crated user

```

```

nc = user_constructor(sourceUser, username, targetFolder)

# List of field names which we cannot copy
do_not_copy = ["id"]

# Duplicate field data from old user object to new one by inspecting the user_
↪object schema
for field in lc.Schema().fields():
    name = field.getName()

    # ComputedFields are handled specially,
    # and UID also
    if not isinstance(field, atapi.ComputedField) and name not in do_not_copy:

        if not field.writeable(nc):
            raise RuntimeError("No permission to copy field value:" + name)

        if name == "password":
            # Note: moving password from one user to another
            # is not possible because password is hashed with
            # the user location in Products.remember.content.password_hashers
            # Insert dummy password which must be reseted
            nc.password = "dummy"
        else:
            value = field.getRaw(lc)

            # The schema of new object
            schema = nc.Schema()

            # Check that the old field exists in the new schema
            if name in schema:
                newfield = schema[name]
                logger.debug("Copying field " + name + " " + str(value))
                newfield.set(nc, value)
            else:
                # The old field does not exist on the new object
                logger.warning("Target does not have field " + name)

# Do custom setup for newly created user
post_process(lc, nc)

# Validate NC user
errors = {}
nc.Schema().validate(nc, None, errors, True, True)
if errors:
    assert not errors, "Newly created user did not validate:" + str(errors)

# Assert that the user is not yet log in-able
workflow = getToolByName(lc, "portal_workflow")
review_state = workflow.getInfoFor(nc, 'review_state')
assert review_state == expected_creation_state, "Got review state:" + review_state

# Remove the old user object
parent = lc.aq_parent

##fore email-catalog removal and without the -old added
lc_path='/'.join(lc.getPhysicalPath()).replace('-old','')
parent.manage_delObjects([lc.getId()])

```

```

# Trigger workflow state transition to register
# Mark creation flag to be set

nc.markCreationFlag()

assert nc.isValid(), "The new NC was not valid after the creation flag was set"

# This will trigger automatic workflow transition
# to the registered state
nc.update()

# Validate NC user once again, just in case markCreationFlag and update did
↳ something bad
errors = {}
nc.Schema().validate(nc, None, errors, True, True)
if errors:
    assert not errors, "Got errors:" + str(errors)
nc.reindexObject()

# Check if we have betahaus.emailcatalog extension installed for Plone 3.x
email_catalog = getToolByName(nc, "email_catalog", default=None)

if email_catalog is not None:
    # This ensures the member log-in will work in the future
    # as email_catalog does not automatically reflect member changes
    email_catalog.uncatalog_object(lc_path)
    email_catalog.reindexObject(nc)

# Not needed - this email is automatically triggered by
# workflow state change when the all user fields are
# validated successfully in Schema()
#nc.resetPassword()

# Check that we are in active user state - the registration email should have
↳ been send
review_state = workflow.getInfoFor(nc, 'review_state')
assert review_state == expected_initialization_state, "Newly created user was not
↳ auto-activated for some reason, state:" + review_state

return nc

```

Configuring default roles with Dexterity

To configure default roles for Dexterity-based members, you need a class providing the `IMembraneUserRoles` interface, and to register it as adapter.

Define the class (here, in a file named `roles.py`):

```

from Products.membrane.interfaces import IMembraneUserRoles
from dexterity.membrane.behavior.membraneuser import DxUserObject
from dexterity.membrane.behavior.membraneuser import IMembraneUser
from zope.component import adapter
from zope.interface import implementer

```

```
DEFAULT_ROLES = ['Member']

@implementer(IMembraneUserRoles)
@adapter(IMembraneUser)
class MyDefaultRoles(DxUserObject):

    def getRolesForPrincipal(self, principal, request=None):
        return DEFAULT_ROLES
```

And register this class in `configure.zcml`:

```
<adapter
    factory=".roles.MyDefaultRoles"
    provides="Products.membrane.interfaces.IMembraneUserRoles"
/>
```

Sharing

TODO: remove this file, eventually move code example to a “cookbook” section.

Warning: Out of date

This page is out of date. Please visit: [Local Roles](#).

Description

Customizing the sharing feature of Plone

Introduction

- [Sharing tab source code](#)
- [Default sharing tab role translations](#)
- <https://pypi.python.org/pypi/collective.sharingroles>
- <http://encolpe.wordpress.com/2010/02/08/add-a-new-role-in-the-sharing-tab-for-plone-3/>

Setting sharing rights programmatically

Complex example: Create one folder per group and add sharing rights

The sample code

- Creates one folder per group, with some groups excluded. The folder is not created if it exists.
- Blocks role inheritance for the group
- Gives edit access to the group through sharing

- Gives view access to the logged in users through sharing

Example is provided as Zope External Method. Create External Method in the target parent folder through the Management Interface. Then run “Test” for this external method in the Management Interface.

```
import traceback
from StringIO import StringIO
from zope.component import getUtility
from plone.i18n.normalizer.interfaces import IURLNormalizer

block_groups = ["Administrators", "opettajat", "kouluttajat", "yhteyshenkilot"]

def set_sharing(self):

    try:
        buffer = StringIO()
        context = self
        normalizer = getUtility(IURLNormalizer)

        site = context.portal_url.getPortalObject()
        acl = site.acl_users
        groups = acl.source_groups.getGroupIds()

        existing_folders = context.objectIds()

        # Create a folder per each group
        for g in groups:

            if g in block_groups:
                continue

            print >> buffer, "Doing group:" + g

            g = g.decode("utf-8")

            id = normalizer.normalize(g)
            if not id in existing_folders:
                context.invokeFactory("Folder", id)

            folder = context[id]

            # Set sharing rights
            # - No inheritance
            folder.__ac_local_roles_block__ = True

            # - Group has edit access

            # - Logged in users have view access

        except Exception, e:
            traceback.print_exc(buffer)

    return buffer.getvalue()
```

General methods to manipulate local roles (sharing)

```
folder.manage_setLocalRoles(userid, ['Reader'])
```

would grant the role “Reader” (Can View on the Sharing Tab) to `userid`.

Beware that this will set the local roles for the user to only ['Reader']. If the user already has other local roles, this will (untested) clear those.

It will not affect inherited roles.

Security

Zope provides various built-in security facilities

- User - role - permission three layer security model
- Security declarations in ZCML for views, adapters, etc.
- RestrictedPython to evaluate sandboxed code

Programming Guidelines

Use of structure in page templates

Do not use the *structure* page template feature with unfiltered input. This can lead to XSS attacks.

Example potentially dangerous page template snippet:

```
<figcaption tal:content="structure context/image_caption" />
```

The reason this snippet is dangerous is because *image_caption* is not filtered input/output.

Manual filtering

If you need to use *structure* with unfiltered input, you can manually run Plone’s output filtering engine on arbitrary html.

Example:

```
from plone import api

def get_safe_html(context):
    transforms = api.portal.get_tool('portal_transforms')
    data = transforms.convertTo('text/x-html-safe', context.image_caption, mimetype=
    ↪ 'text/html',
                                context=context)

    return data.getData()
```

Persistent traversable object methods

Any persistent objects that can be traversed to (by accessing them via URL), needs to have careful consideration when writing methods.

A legacy artifact of Zope2 is that it automatically published methods with doc strings. If you forgot to specify permissions on a persistent traversable object, and it does writes on the database or discloses information it should not, you could be causing a security issue.

Do not:

```
class MyContent(DexterityItem):
    def foofoo(self):
        """My doc string. This method is public!"""
        self.x = 'foofoo' # mutation
```

Do this:

```
class MyContent(DexterityItem):
    def foofoo(self):
        # My doc string.
        self.x = 'foofoo' # mutation
```

Or this:

```
class MyContent(DexterityItem):
    security = ClassSecurityInfo()
    security.declarePrivate('foofoo')
    def foofoo(self):
        """My doc string. This method is public!"""
        self.x = 'foofoo' # mutation
```

Untrusted JavaScript Input

This isn't a Plone specific guideline, this is for ALL JavaScript people write anywhere.

If you build HTML from user input, make sure to always escape the input.

A common pitfall in jQuery will look like this:

```
var value = '<script>alert("hi")</script>';
$('body').append($(value));
```

By default, jQuery is not safe. To do the previous example in jQuery, you could:

```
var $el = $('<div/>');
var value = '<script>alert("hi")</script>';
$el.text(value);
$('body').append($el);
```

In underscorejs templates make sure to use:

```
<%- ... %>
```

Do not(underscorejs):

```
<%= ... %>
```

Other considerations

Many modern frameworks are safe by default. For example, it is difficult to render untrusted, raw HTML in the ReactJS framework.

Permissions

Description

How to deal with permissions making your code permission-aware in Plone

Introduction

Permissions control whether logged-in or anonymous users can execute code and access content.

Permissions in Plone are managed by Zope's `AccessControl` module. Persistent permission setting and getting by role heavy lifting is done by `AccessControl.rolemanager.RoleManager`.

Permission checks are done for:

- every view/method which is hit by incoming HTTP request (Plone automatically publishes traversable methods over HTTP);
- every called method for *RestrictedPython scripts*.

The basic way of dealing with permissions is setting the `permission` attribute of view declaration. For more information see *views*.

Debugging permission errors: Verbose Security

You can turn on `verbose-security` option in buildout to get better traceback info when you encounter a permission problem on the site (you are presented a login dialog).

For the security reasons, this option is disabled by default.

- Set `verbose-security = on` in your `buildout.cfg` instance or related section.
- Rerun buildout
- Restart Plone properly after buildout `bin/plonectl stop && bin/plonectl start`
- remove the `Unauthorized` exception from the list of ignored exceptions inside the `error_log` object within the Plone root folder through the Management Interface

More info

- <https://pypi.python.org/pypi/plone.recipe.zope2instance>

Checking if the logged-in user has a permission

The following code checks whether the logged in user has a certain permission for some object.


```

from AccessControl import getSecurityManager
from AccessControl import Unauthorized

# Import permission names as pseudo-constant strings from somewhere...
# see security doc for more info
from Products.CMFCore.permissions import ModifyPortalContent

def some_function(self, obj):
    sm = getSecurityManager()
    if not sm.checkPermission(ModifyPortalContent, obj):
        raise Unauthorized("You need ModifyPortalContent permission to execute some_
↪function")

    # ...
    # we have security clearance here
    #

```

Checking whether a specific role has a permission

The following example uses the `rolesOfPermission()` method to check whether the *Authenticated* role has a permission on a certain folder on the site. The weirdness of the method interface is explained by the fact that it was written for use in a Management Interface template:

```

def checkDBPermission(self):
    from zope.app.component.hooks import getSite
    site = getSite()
    obj = site.intranet
    perms = obj.rolesOfPermission("View")
    found = False

    for perm in perms:
        if perm["name"] == "Authenticated":
            if perm["selected"] != "": # will be SELECTED if the permission is granted
                found = True
                break

    if not found:
        from Products.statusmessages.interfaces import IStatusMessage
        messages = IStatusMessage(self.request)
        messages.addStatusMessage(u"Possible permission access problem with the_
↪intranet. Errors on creation form may happen.", type="info")

```

Permission Access

Objects that are manageable *TTW* inherit from `RoleManager`. The API provided by this class permits you to manage permissions.

Example: see all possible permissions:

```

>>> obj.possible_permissions()
['ATContentTypes Topic: Add ATBooleanCriterion',
 'ATContentTypes Topic: Add ATCurrentAuthorCriterion',
 ...
]

```

Show the security matrix of permission:

```
>>> self.portal.rolesOfPermission('Modify portal content')
[{'selected': '', 'name': 'Anonymous'},
 {'selected': '', 'name': 'Authenticated'},
 {'selected': '', 'name': 'Contributor'},
 {'selected': '', 'name': 'Editor'},
 {'selected': 'SELECTED', 'name': 'GroupAdmin'},
 {'selected': '', 'name': 'GroupContributor'},
 {'selected': '', 'name': 'GroupEditor'},
 {'selected': '', 'name': 'GroupLeader'},
 {'selected': '', 'name': 'GroupMember'},
 {'selected': '', 'name': 'GroupReader'},
 {'selected': '', 'name': 'GroupVisitor'},
 {'selected': 'SELECTED', 'name': 'Manager'},
 {'selected': '', 'name': 'Member'},
 {'selected': 'SELECTED', 'name': 'Owner'},
 {'selected': '', 'name': 'Reader'},
 {'selected': '', 'name': 'Reviewer'},
 {'selected': '', 'name': 'SubscriptionViewer'}]
```

Bypassing permission checks

The current user is defined by active security manager. During both restricted and unrestricted execution certain functions may do their own security checks (`invokeFactory`, `workflow`, `search`) to filter out results.

If a function does its own security checks, there is usually a code path that will execute without security check. For example the methods below have security-aware and raw versions:

- `context.restrictedTraverse()` vs. `context.unrestrictedTraverse()`
- `portal_catalog.searchResults()` vs. `portal_catalog.unrestrictedSearchResults()`

However, in certain situations you have only a security-aware code path which is blocked for the current user. You still want to execute this code path and you are sure that it does not violate your site security principles.

Below is an example how you can call any Python function and work around the security checks by establishing a temporary `AccessControl.SecurityManager` with a special role.

Example:

```
from AccessControl import ClassSecurityInfo, getSecurityManager
from AccessControl.SecurityManagement import newSecurityManager, setSecurityManager
from AccessControl.User import nobody
from AccessControl.User import UnrestrictedUser as BaseUnrestrictedUser

class UnrestrictedUser(BaseUnrestrictedUser):
    """Unrestricted user that still has an id.
    """
    def getId(self):
        """Return the ID of the user.
        """
        return self.getUserName()

def execute_under_special_role(portal, role, function, *args, **kwargs):
    """ Execute code under special role privileges.

    Example how to call::
```

```

        execute_under_special_role(portal, "Manager",
                                   doSomeNormallyNotAllowedStuff,
                                   source_folder, target_folder)

@param portal: Reference to ISiteRoot object whose access controls we are using
@param function: Method to be called with special privileges

@param role: User role for the security context when calling the privileged code;
→e.g. "Manager".

@param args: Passed to the function

@param kwargs: Passed to the function
"""

sm = getSecurityManager()

try:
    try:
        # Clone the current user and assign a new role.
        # Note that the username (getId()) is left in exception
        # tracebacks in the error_log,
        # so it is an important thing to store.
        tmp_user = UnrestrictedUser(
            sm.getUser().getId(), '', [role], ''
        )

        # Wrap the user in the acquisition context of the portal
        tmp_user = tmp_user.__of__(portal.acl_users)
        newSecurityManager(None, tmp_user)

        # Call the function
        return function(*args, **kwargs)

    except:
        # If special exception handlers are needed, run them here
        raise
finally:
    # Restore the old security manager
    setSecurityManager(sm)

```

For a more complete implementation of this technique, see:

- http://github.com/ned14/Easyshop/blob/master/src/easyshop.order/easyshop/order/adapters/order_management.py

Catching Unauthorized

Gracefully failing when the user does not have a permission. Example:

```

from AccessControl import Unauthorized

try:
    portal_state = context.restrictedTraverse("@@plone_portal_state")
except Unauthorized:

```

```
# portal_state may be limited to admin users only
portal_state = None
```

Creating permissions

Permissions are created declaratively in *ZCML*. Before Zope 2.12 (that is, before Plone 4), the `collective.autopermission` package was required to enable this, but now it is standard behaviour.

- <http://n2.nabble.com/creating-and-using-your-own-permissions-in-Plone-3-tp339972p1498626.html>
- <http://blog.fourdigits.nl/adding-zope-2-permissions-using-just-zcml-and-a-generic-setup-profile>

Example:

```
<configure
  xmlns="http://namespaces.zope.org/zope"
  xmlns:browser="http://namespaces.zope.org/browser">

  <include package="collective.autopermission" />

  <permission
    id="myproduct.mypermission"
    title="MyProduct: MyPermission"
  />

  <browser:page
    for="*"
    name="myexampleview"
    class="browser.MyExampleView"
    permission="myproduct.mypermission"
  />

</configure>
```

Now you can use the permission both as a Zope 2-style permission (`MyProduct: MyPermission`) or a Zope 3-style permission (`myproduct.mypermission`). The only disadvantage is that you can't import the permission string as a variable from a `permissions.py` file, as you can with permissions defined programmatically.

By convention, the permission id is prefixed with the name of the package it's defined in, and uses lowercase only. You have to take care that the title matches the permission string you used in `permissions.py` exactly — otherwise a different, Zope 3 only, permission is registered.

Zope 3 style permissions are necessary when using Zope 3 technologies such as `BrowserViews`/`formlib`/`z3c.form`. For example, from `configure.zcml`:

```
<configure
  xmlns="http://namespaces.zope.org/zope"
  xmlns:browser="http://namespaces.zope.org/browser">

  <permission
    id="myproduct.mypermission"
    title="MyProduct: MyPermission" />

  <browser:page
    for="*"
    name="myexampleview"
    class="browser.MyExampleView"
```

```

    permission="myproduct.mypermission"
  />

</configure>

```

Define Zope 2 permissions in Python code (old style)

If you want to protect certain actions in your product by a special permission, you most likely will want to assign this permission to a role when the product is installed. You will want to use Generic Setup's `rolemap.xml` to assign these permissions. A new permission will be added to the Zope instance by calling `setDefaultRoles` on it.

However, at the time when Generic Setup is run, almost none of your code has actually been run, so the permission doesn't exist yet. That's why we define the permissions in `permissions.py`, and call this from `__init__.py`:

`__init__.py`:

```
import permissions
```

`permissions.py`:

```

from Products.CMFCore import permissions as CMFCorePermissions
from AccessControl.SecurityInfo import ModuleSecurityInfo
from Products.CMFCore.permissions import setDefaultRoles

security = ModuleSecurityInfo('MyProduct')
security.declarePublic('MyPermission')
MyPermission = 'MyProduct: MyPermission'
setDefaultRoles(MyPermission, ())

```

When working with permissions, always use the variable name instead of the string value. This ensures that you can't make typos with the string value, which are hard to debug. If you do make a typo in the variable name, you'll get an `ImportError` or `NameError`.

Assigning permissions to users (roles)

Permissions are usually assigned to roles, which are assigned to users through the web.

To assign a permission to a role, use `profiles/default/rolemap.xml`:

```

<?xml version="1.0"?>
<rolemap>
  <permissions>
    <permission name="MyProduct: MyPermission" acquire="False">
      <role name="Member"/>
    </permission>
  </permissions>
</rolemap>

```

Manually fix permission problems

In the case you fiddle with permission and manage to lock out even the admin user you can still fix the problem from the *debug prompt*.

Example debug session, restoring Access Contents Information for all users:

```
>>> c = app.yoursiteid.yourfolderid.problematiccontent
>>> import AccessControl
>>> from Products.CMFCore.permissions import AccessContentsInformation
>>> sm = AccessControl.getSecurityManager()
>>> import transaction
>>> anon = sm.getUser()
>>> c.manage_permission(AccessContentsInformation, roles=anon.getRoles())
>>> transaction.commit()
```

Available permissions in Plone

Description

What Zope security permissions you have available for your Plone coding

Listing different available permissions

Each permission name is a string.

To see available permissions, click Security tab at your site root in the Management Interface.

In programming, use pseudoconstants instead of permission string values:

- See `CMFCore.permissions`
- See `AccessControl.Permissions`

For available ZCML permission mappings see:

- `Products/Five/permissions.zcml`
 - Permissions such as `cmf.ModifyPortalContent`, `zope2.View`
- `zope/security/permissions.zcml`
 - `zope.Public`

or search for the string `<permission in *.zcml` files in the *eggs* folder of your Plone development deployment.

Example using UNIX `grep` tool:

```
grep -C 3 -Ri --include=*.zcml "<permission" *
```

Useful permissions

Permissions are shown by their verbose name in the Management Interface.

View This governs whether you are allowed to view some content.

Access Contents Information This permission allows access to an object, without necessarily viewing the object. For example, a user may want to see the object's title in a list of results, even though the user can't view the contents of that file.

List folder contents This governs whether you can get a listing of the contents of a folder; it doesn't check whether you have the right to view the objects listed.

Modify Portal Content This governs whether you are allowed to modify some content.

Manage Portal This permission allows you to manage the portal. A number of views in the plone control panel are protected with this view. If you plan to write a reusable product, be very hesitant to use this permission, check whether a custom permission might make more sense.

There is no single permission for adding content. Every content type has its own permission. If you create your own content type, create a custom add permission for it.

Table 6.1: Permissions

Permission name	Permission name for ZCML
View	zope2.View
Access contents information	zope2.AccessContentsInformation
List folder contents	cmf.ListFolderContents
Modify portal content	cmf.ModifyPortalContent
Manage portal	cmf.ManagePortal

To reference a permission in code, you need the name as a string. Using strings is a bad convention, all common permissions have a constant in `Products.CMFCore.permissions`. To perform a permission check properly, you do something like this:

```
from AccessControl import getSecurityManager
from AccessControl import Unauthorized
from Products.CMFCore import permissions

if not getSecurityManager().checkPermission(permissions.ModifyPortalContent, object):
    raise Unauthorized("You may not modify this object")
```

All standard permissions from above can be referenced by their Permission name without spaces.

More info:

- <http://markmail.org/thread/3izsoh2ligthfcou>

Standard permissions and roles

Description

Technical overview of Plones standard permissions and roles.

Standard permissions

The standard permissions can be found in `AccessControl`'s and `Product.CMFCore`'s `permissions.zcml`. Here, you will find a short id (also known as the *Zope 3 permission id*) and a longer title (also known as the *Zope 2 permission title*). For historical reasons, some areas in Plone use the id, whilst others use the title. As a rule of thumb:

- Browser views defined in ZCML directive use the Zope 3 permission id;
- Security checks using `zope.security.checkPermission()` use the Zope 3 permission id;
- Dexterity's `add_permission` FTI variable uses the Zope 3 permission id;
- The `rolemap.xml` GenericSetup handler and workflows use the Zope 2 permission title;
- Security checks using `AccessControl`'s `getSecurityManager().checkPermission()`, including the methods on the `portal_membership` tool, use the Zope 2 permission title.

The most commonly used permission are shown below. The Zope 2 permission title is shown in parentheses.

zope2.View (*View*) used to control access to the standard view of a content item;

zope2.DeleteObjects (*Delete objects*) used to control the ability to delete child objects in a container;

cmf.ModifyPortalContent (*Modify portal content*) used to control write access to content items;

cmf.ManagePortal (*Manage portal*) used to control access to management screens;

cmf.AddPortalContent (*Add portal content*) the standard add permission required to add content to a folder;

cmf.SetOwnProperties (*Set own properties*) used to allow users to set their own member properties'

cmf.RequestReview (*Request review*) typically used as a workflow transition guard to allow users to submit content for review;

cmf.ReviewPortalContent (*Review portal content*) usually granted to the `Reviewer` role, controlling the ability to publish or reject content.

Standard roles

As with permissions, it is easy to create custom roles (use the `rolemap.xml` `GenericSetup` import step – see CMFPlone's version of this file for an example), although you should use the standard roles where possible.

The standard roles in Plone are:

Anonymous a pseudo-role that represents non-logged in users.

Note: if a permission is granted to *Anonymous*, it is effectively granted to everyone. It is not possible to grant permissions to non-logged in users without also granting them to logged in ones.

Authenticated a pseudo-role that represents logged-in users.

Owner automatically granted to the creator of an object.

Manager which represents super-users/administrators. Almost all permissions that are not granted to *Anonymous* are granted to *Manager*.

Site Manager which represents site/administrators. Has permissions needed to fully manage a single Plone site.

Reviewer which represents content reviewers separately from site administrators. It is possible to grant the *Reviewer* role locally on the *Sharing* tab, where it is shown as *Can review*.

Member representing “standard” Plone users.

In addition, there are three roles that are intended to be used as *local roles* only. These are granted to specific users or groups via the *Sharing* tab, where they appear under more user friendly pseudonyms.

Reader, aka Can view, confers the right to view content. As a rule of thumb, the *Reader* role should have the *View* and *Access contents information* permissions if the *Owner* roles does.

Editor, aka Can edit, confers the right to edit content. As a rule of thumb, the *Editor* role should have the *Modify portal content* permission if the *Owner* roles does.

Contributor, aka Can add, confers the right to add new content. As a rule of thumb, the:guilabel: *Contributor* role should have the *Add:guilabel: portal content* permission and any type-specific add permissions globally (i.e. granted in `rolemap.xml`), although these permissions are sometimes managed in workflow as well.

Custom permissions

Description

Creating special permissions for your product

If you want to protect certain actions in your product by a special permission, you most likely will want to assign this permission to a role when the product is installed.

First the permission is defined in *zcml*. It includes an example how to use the permission in a browser page

```
<configure
  xmlns="http://namespaces.zope.org/zope"
  xmlns:browser="http://namespaces.zope.org/browser">

  <permission
    id="myproduct.mypermission"
    title="MyProduct: MyPermission"
  />

  <browser:page
    for="*"
    name="myexampleview"
    class="browser.MyExampleView"
    permission="myproduct.mypermission"
  />

</configure>
```

Now you can use the permission both as a Zope 2 permission (*'MyProduct: MyPermission'*) or a Zope 3 permission (*'myproduct.mypermission'*). The only disadvantage is that you can't import the permissionstring as a variable from a *permissions.py* like from *Products.CMFCore.permissions*.

Use Generic Setup's *rolemap.xml* to assign the new permission to roles. This defines the defaults. With the use of (custom) workflows this mapping may change.

```
<?xml version="1.0"?>
<rolemap>
  <permissions>
    <permission name="MyProduct: MyPermission"
      acquire="True">
      <role name="Manager"/>
      <role name="Site Administrator"/>
      <role name="Owner"/>
      <role name="Contributor"/>
    </permission>
  </permissions>
</rolemap>
```

A new permission will be added to the whole Zope instance by calling *setDefaultRoles* on it. This step is only *rarely needed*, i.e. if the permission must be available outside of Plone Site.

Define the following code in your *__init__.py*:

```
from Products.CMFCore.permissions import setDefaultRoles

setDefaultRoles('MyProduct: MyPermission', ('Manager', 'Owner',))
```

Cross-Site Request Forgery (CSRF)

Plain usage

Documentation: <https://github.com/plone/plone.protect/>

z3c.form

z3c.form does not include csrf protection yet: <https://bugs.launchpad.net/z3c.form/+bug/805794>

Local roles

Description

Creating and setting local roles of Plone members programmatically.

Introduction

Local roles allows user accounts to have special privileges for a folder and its children.

By default Plone has roles like Contributor, Reader, Editor, etc. You can view these on the *Sharing* tab and in the Management Interface *Security* tab.

Good introduction to roles: [Basic Roles and Permissions in Plone](#)

Creating a new role

New Plone roles can be created through the *GenericSetup rolemap.xml* file.

Example `profiles/default/rolemap.xml`

```
<?xml version="1.0"?>
<rolemap>
  <roles>
    <role name="National Coordinator"/>
    <role name="Site Manager"/>
  </roles>
  <permissions>
  </permissions>
</rolemap>
```

Adding a role to the Sharing Tab

To let the newly created role appear in the @@sharing tab, create a *GenericSetup sharing.xml* file.

Example `profiles/default/sharing.xml`

```
<sharing xmlns:i18n="http://xml.zope.org/namespaces/i18n"
         i18n:domain="plone">
  <role
    id="Site Manager"
    title="Is a site coordinator"
    permission="Manage portal"
    i18n:attributes="title"
  />
</sharing>
```

The title is the name to be shown on the sharing page. The `required_permission` is optional. If given, the user must have this permission to be allowed to manage the particular role.

Setting local role

`manage_setLocalRoles` is defined in `AccessControl.rolemanager.RoleManager`.

Example:

```
context.manage_setLocalRoles(userid, ["Local roles as a list"])
```

Getting local roles

The `get_local_roles()` method returns currently-set local roles. This does not return all the *effective* roles (which may include roles acquired from the parent hierarchy). `get_local_roles_for_userid()` returns roles for a particular user as a tuple.

Example:

```
# get_local_roles() return sequence like ( ("userid1", ("rolename1", "rolename2")), (
→ "userid2", ("rolename1") )
roles = context.get_local_roles()
```

The `getRolesInContext()` method returns the list of roles assigned to the user, including local roles assigned in the context of the passed-in object and both local roles assigned directly to the user and those assigned to the user's groups.

Example:

```
from Products.CMFCore.utils import getToolByName
pm = getToolByName(context, 'portal_membership')
roles_in_context = pm.getAuthenticatedMember().getRolesInContext(context)
```

Deleting local roles

`manage_delLocalRoles(userids)` takes a *list of usernames* as argument. All local roles for these users will be cleared.

The following example (membrane-specific) will reset local roles based on external input

```
def _updateLocalRoles(self):
    """ Resets Local Coordinator roles for associated users.

    Reads Archetypes field which is a ReferenceField to membrane users.
```

```
Based on this field values users are granted local roles on this object.
"""

# Build list of associated usernames
usernames = []

# Set roles for newly given users
for member in self.getExtraLocalCoordinators():

    # We are only interested in this particular custom membrane user type
    if member.getUserType() == "local_coordinator":

        username = member.getUserName()

        usernames.append(username)

        self.manage_setLocalRoles(username, ["Local Coordinator"])

membrane = getToolByName(self, "membrane_tool")

# Make sure that users which do not appear in extraLocalCoordinators
# will have their roles cleared
for username, roles in self.get_local_roles():

    sits_user = membrane.getUserAuthProvider(username)

    if not username in usernames:
        print "Clearing:" + username
        self.manage_delLocalRoles([username])
```

Blocking local roles

Local roles may need to be blocked on a particular object by default. This can be achieved by add a flag to your content object, like so:

```
class MyType(content.Container):
    """My content type
    """
    implements(IMyType)
    __ac_local_roles_block__ = True
```

Local role caching

Resolving effective local roles is a cumbersome operation, so the result is cached.

Warning: Unit testers: Local roles are cached per request. You need to clear this cache after modifying an object's local roles or switching user if you want to get proper readings.

Unit test example method:

```
def clearLocalRolesCache(self):
    """ Clear borg.localroles cache.
```

```
borg.localroles check role implementation caches user/request combinations.
If we edit the roles for a user we need to clear this cache,
"""
from zope.annotation.interfaces import IAnnotations
ann = IAnnotations(self.app.REQUEST)
for key in list(ann.keys()): # Little destructive here, deletes *all* annotations
    del ann[key]
```

Debugging

Set your breakpoint in `Products.PlonePAS.plugins.local_role.LocalRolesManager.getRolesInContext()` and `Products.PlonePAS.plugins.role.GroupAwareRoleManager.getRolesForPrincipal()`. There you see how roles for a given context are being resolved.

Check the `acl_users.portal_role_manager` via the Management Interface.

Please see the `zopyx.plone.cassandra` add-on product.

Other

- <http://toutpt.wordpress.com/2009/03/14/plone-and-local-roles-too-quiet/>

Dynamic roles

Introduction

Plone core's `borg.localrole` package allows you to hook into role-resolving code and add roles dynamically. I.e. the role on the user depends on HTTP request / environment conditions and is not something set in the site database.

Creating a dynamic role

First *create an Plone add-on for your coding needs*.

`getRoles()` function is called several times per request so you might want to cache the result.

There is a complex example below.

- `getAllRoles()` is overridden to return a custom role which is not available through normal security machinery. This is required because Plone/Zope builds look-up tables based on the result of `getAllRoles()` and all possible roles must appear there
- `getRoles()` is overridden to call custom `getDummyRolesOnContext()` which has the actual logic to resolve the roles
- An example code checks whether the context object implements a marker interface and gives the user a role based on that

Example `localroles.py`:

```
from zope.interface import Interface, implements
from zope.component import adapts
from zope.component.interfaces import ISiteManager
from borg.localrole.interfaces import ILocalRoleProvider
```

```

from plone.memoize import forever
from Products.CMFCore.utils import getToolByName
from Products.DummyHospital.interfaces import IDummyHospital, IDummyCountry

class DummyLocalRoleAdapter(object):
    """ Give additional Member roles based on context and DummyUser type.

    This enables giving View permission on items higher in the
    traversign path than the user folder itself.
    """
    implements(ILocalRoleProvider)
    adapts(Interface)

    def __init__(self, context):
        self.context = context

    def getDummyRolesOnContext(self, context, principal_id):
        """ Calculate magical Dummy roles based on the user object.

        Note: This function is *heavy* since it wakes lots of objects along the
        ↪ acquisition chain.
        """

        # Filter out bogus look-ups - Plone calls this function
        # for every possible role look up out there, but
        # we are interested only these two cases
        if IDummyMarkerInterface.providedBy(context):
            return ["Dummy Member"]

        # No match
        return []

    def getRoles(self, principal_id):
        """Returns the roles for the given principal in context.

        This function is additional besides other ILocalRoleProvider plug-ins.

        @param context: Any Plone object
        @param principal_id: User login id
        """
        return self.getDummyRolesOnContext(self.context, principal_id)

    def getAllRoles(self):
        """Returns all the local roles assigned in this context:
        (principal_id, [role1, role2])"""
        return [ ("dummy_id", ["Dummy Member"]) ]

```

Custom local role implementation is made effective using ZCML adapter directive in your add-ons configure.
zcml:

```

<configure
  xmlns="http://namespaces.zope.org/zope"
  xmlns:zcml="http://namespaces.zope.org/zcml">

  <include package="borg.localrole" />

```

```
<adapter
  factory=".localroles.DummyLocalRoleAdapter"
  name="dummy_local_role"
/>

</configure>
```

If your dynamic role is not any of Plone’s existing roles you need to *declare it with `rolemap.xml`*.

Sandboxing and RestrictedPython

Description

Legacy Plone code uses RestrictedPython sandboxing to secure each module and class functions.

Introduction

Plone has two sandboxing modes

- Unrestricted: Python code is executed normally and the code can access the full Zope application server environment. This includes other site instances too. This is generally what happens when you write your own add-on and add views for it.
- Restricted (RestrictedPython): scripts and evaluations are specially compiled, have limited Python language functionality and every function call is checked against the security manager. This is what happens when you try to add Python code or customize page templates through the Management Interface.

Restricted execution is enabled only for **through-the-web** scripts and **legacy code**:

- Old style TAL page templates: everything you put inside page template `tal:content`, `tal:condition`, etc. These templates are `.pt` templates **without** accompanying `BrowserView`
- Script (Python) code is executed (`plone_skins` layer Python scripts and old style form management)

Note: RestrictedPython was bad idea and mostly causes headache. Avoid through-the-web Zope scripts if possible.

For further information, read

- <http://plone.293351.n2.nabble.com/Update-was-Plone-4-Chameleon-compatibility-tp5612838p5614466.html>

Whitelisting modules for RestrictedPython import

- <https://plone.org/documentation/kb/using-unauthorized-modules-in-scripts>

Traversing special cases

Old style Zope object traversing mechanism does not expose

- Functions without docstring (the `“””` comment at the beginning of the function)
- Functions whose name begins with underscore (`“_”`-character)

Unit testing RestrictedPython code

RestrictedPython code is problematic, because RestrictedPython hardening is done on Abstract Syntax Tree level and effectively means all evaluated code must be available in the source code form. This makes testing RestrictedPython code little difficult.

Below are few useful unit test functions:

```
# Zope security imports
from AccessControl import getSecurityManager
from AccessControl.SecurityManagement import newSecurityManager
from AccessControl.SecurityManagement import noSecurityManager
from AccessControl.SecurityManager import setSecurityPolicy
from AccessControl import ZopeGuards
from AccessControl.ZopeGuards import guarded_getattr, get_safe_globals, safe_builtins
from AccessControl.ImplPython import ZopeSecurityPolicy
from AccessControl import Unauthorized

# Restricted Python imports
from RestrictedPython import compile_restricted
from RestrictedPython.SafeMapping import SafeMapping

def _execUntrusted(self, debug, function_body, **kwargs):
    """ Sets up a sandboxed Python environment with Zope security in place.

    Calls func() in an sandboxed environment. The security mechanism
    should catch all unauthorized function calls (declared
    with a class SecurityManager).

    Security is effective only inside the function itself -
    The function security declarations themselves are ignored.

    @param func: Function object
    @param args: Parameters delivered to func
    @param kwargs: Parameters delivered to func
    @param debug: If True, break into pdb debugger just before evaluation
    @return: Function return value
    """

    # Create global variable environment for the sandbox
    globals = get_safe_globals()
    globals['__builtins__'] = safe_builtins

    # Zope seems to have some hacks with guaded_getattr.
    # guarded_getattr is used to check the permission when the
    # object is being traversed in the restricted code.
    # E.g. this controls function call permissions.
    from AccessControl.ImplPython import guarded_getattr as guarded_getattr_safe
    globals['_getattr_'] = guarded_getattr_safe
    #globals['getattr'] = guarded_getattr_safe
    #globals['guarded_getattr'] = guarded_getattr_safe

    globals.update(kwargs)

    # Our magic code

    # The following will compile the parsed Python code
```



```

# and applies a special AST mutator
# which will proxy __getattr__ and function calls
# through guarded_getattr
code = compile_restricted(function_body, "<string>", "eval")

# Here is a good place to break in
# if you need to do some ugly permission debugging
if debug:
    pass # go pdb here

return eval(code, globals)

def execUntrusted(self, func, **kwargs):
    """ Sets up a sandboxed Python environment with Zope security in place. """
    return self._execUntrusted(False, func, **kwargs)

def execUntrustedDebug(self, func, **kwargs):
    """ Sets up a sandboxed Python debug environment with Zope security in place. """
    return self._execUntrusted(True, func, **kwargs)

def assertUnauthorized(self, func, **kwargs):
    """ Check that calling func with currently effective roles will raise
    ↳ Unauthorized error. """
    try:
        self.execUntrusted(func, **kwargs)
    except Unauthorized, e:
        return

    raise AssertionError, 'Unauthorized exception was expected'

def test_xxx(self):
    # Run RestrictedPython in unit test code
    # myCustomUserCreationFunction() is view/Python script/method you must call in
    ↳ the restricted mode
    self.execUntrusted('portal.myCustomUserCreationFunction(username="national_
    ↳ coordinator", email="nationalcoordinator@redinnovation.com")', portal=self.portal)

```

Other references

- [zope.security](#)

Using SELinux with Plone

Description

Tutorial on using SELinux with Plone, using Plone 4.3 and RedHat Linux 6.3.

Introduction

This document is a tutorial on using SELinux with Plone, using RedHat Linux 6.3 and Plone 4.3. It is applicable to any Linux distribution with small changes.

About SELinux

SELinux is a mandatory access control system, meaning that SELinux assigns security *contexts* (presented by *labels*) to system resources, and allows access only to the processes that have defined required levels of authorization to the contexts. In other words, SELinux maintains that certain *target* executables (having security contexts) can access (level of access being defined explicitly) only certain files (having again security context labels). In essence the contexts are roles, which makes SELinux a Role Based Access Control system. It should be noted that even root is usually just an ordinary user for RBAC systems, and will be contained like any other user.

The concept of contexts and labels can be slightly confusing at first. It stems from the idea of chain of trust. A system that upholds that proper authorization checks are being done is worthless if the system allows moving the protected data to a place that does not have similar authorization checks. Context labels are file system attributes, and when the file is moved around the label (representing context) moves with the file. The system is supposed to limit where the information can be moved, and the contexts can be extended beyond file system (ie. labels on rows in database systems), building complete information systems that will never hand over data to a party that is unable (or unwilling) to take care of it.

Most SELinux policies *target* an executable, and define the contexts (usually applied with labels to files) it can access by using *type enforcement rules*. However there are also *capabilities* that control more advanced features such as the ability to execute heap or stack, setuid, fork process, bind into ports, or open TCP sockets. Most of the capabilities and macros come from reference policy, which offers policy developers ready solutions to most common problems. The reference policy shipped by Linux distributions contains ready rules for some 350 targets, including applications like most common daemons (sshd), and system services (init/systemd).

The value of SELinux is in giving administrators fine granularity of access control far beyond the usual capabilities of *NIX systems. This is useful especially in mitigating the impact of security vulnerabilities. The most apparent downside to SELinux is the high skill requirements. To understand most of SELinux - and to be able to maintain it effectively with 3rd party applications - requires good abstraction skills, and especially the official documentation is somewhat hard to digest. SELinux was never engineered to be easy for administrators. It was engineered to be able to implement complex security models like Bell-LaPadula and MLS.

There have been several myths about SELinux being heavy (in reality it comes with ~3% overhead), or that it breaks all applications. There used to be time (years ago) when SELinux applied itself by default on everything, and if the application was not included in the shipped policies it probably failed miserably. Most of the application developers and companies got frustrated to the situation, and started recommending that SELinux should always be disabled. Things have luckily changed drastically since then. Today most SELinux implementations use what is called *targeted policy*, which means that SELinux affects only applications that have explicit policies. As a result SELinux does generally nothing to your 3rd party applications - good or bad - until you enable it. This tutorial is meant to give readers pointers on how to accomplish exactly that.

Creating new SELinux policy

Prerequisites

- root access
- Working SELinux (`sudo sestatus` reports **ENABLED**, and **enforcing**)
- Preferably a system that uses *targeted policy* (see the output of previous command)
- SELinux policy utilities installed (policycoreutils-python policycoreutils-gui)
- The application (in this case Plone) already installed

Creating new policy

Development starts usually by generating a policy skeleton with the *sepolgen* (or *sepolicy-generate*) utility. It can generate several types of templates, which come with a set of basic access rights. There are several *sepolgen* versions out there, depending on the Linux distribution. The most important differences between them are in the included templates. Creating new policy is done with the following command:

```
sepolgen -n plone -t 3 /usr/local/Plone/zinstance/bin/plonectl
```

Where the parameters are:

- **-n plone** gives the new policy name. Default is to use the name of the executable, but we want to give a more generic name in this case.
- **-t 3** elects a template (“*normal application*”) that gives some commonly required access rights as a starting point
- **/usr/local/Plone/zinstance/bin/plonectl** is the application that will get a new context (*plonectl_exec_t*), which will get most of the type enforcement rules.

The outcoming result will be four files:

- **plone.te** Type enforcement file defining the access rules. **This file contains most of the policy, and most of the rules go there.**
- **plone.if** Interface file defining what *other* policies can import from your policy.
- **plone.fc** File contexts file defining what context labels will be applied to files and directories.
- **plone.sh** Setup script that will compile and install the policy to the system configuration (both running and persistent).

Labeling files

Before the actual development will start file context labeling rules should be defined in **plone.fc**. You probably need some context (*plone_t*) for all files related to Plone, context (*plone_rw_t*) with write rights to *var* and the *plonectl* will need a context (*plonectl_exec_t*) that comes with special rights.

```
/usr/local/Plone(.*) gen_context(system_u:object_r:plone_t,s0)
/usr/local/Plone/zinstance/var(.*) gen_context(system_u:object_r:plone_rw_t,s0)
/usr/local/Plone/zinstance/bin/plonectl gen_context(system_u:object_r:plonectl_exec_t,
↪s0)
```

The generated **plone.te** already tells SELinux what *plone_t* and *plone_exec_t* are - valid file context types. The tools labeling files will know what to do about them. However the *plone_rw_t* is must be introduced before continuing, and the *plone_t* should be renamed to *plonectl_t* (to describe the target better - important for managing more complex rules):

```
type plonectl_exec_t;
application_domain(plone_t, plonectl_exec_t)
type plone_rw_t;
files_type(plone_rw_t)
```

It is also a good idea to edit the *restorecon* commands at the end of **plone.sh** to point to */usr/local/Plone* and relabel all the files when the policy is recompiled and installed:

```
/sbin/restorecon -F -R -v /usr/local/Plone
```

Development process

The basic policy development process for SELinux policies follows the following pattern:

1. Add permissive rules
2. Compile & install your policy
3. Clear the audit logs
4. Run the application until it fails
5. Run audit2allow
6. Study the output of audit2allow, and add more access rules to satisfy the application
7. Repeat from step 2 until everything works
8. Remove permissive rules

Permissive rules

Most applications require largish amount of rules just to start properly. To reach a working set of rules faster you can switch your contexts to permissive mode by editing the *PlonePython.te*:

```
require {
    type unconfined_t;
}

permissive plone_t;
permissive plonectl_exec_t;
permissive plone_rw_t;
```

Permissive in SELinux means that all actions by mentioned contexts will be allowed to process, and the incidents (*access vector denials*) will be only logged. This will allows to gather rules faster than going through the complete development cycle.

Warning: Please note that permissive rules have to be removed at some point, or the policy will **not** protect the application as expected.

Using audit2allow

Audit2allow can search both dmesg and the system audit logs for access vector cache denials, and build suggestions based on them. Because the output will be more understandable without extra noise, it is recommendable to clear audit log between development cycles. Since it is probably not a good idea to clear dmesg, it is suggested that you clear the system audit logs, and instruct audit2allow to use them as source, for example:

```
cat /dev/null > /var/log/audit.log
# Break the application
audit2allow -r -R -i /var/log/audit/audit.log
```

There are couple useful parameters for running audit2allow:

- *-r* adds requires (“imports” from other policies) to the output

- `-R` makes `audit2allow` suggest compatible macros from other available policies. Macros contain often more lenient access rules, but they also reduce the amount of required rules. Using them will make the policy slightly more platform dependent, but easier to maintain.
- `-i /var/log/audit/audit.log` makes only to audit logs to be evaluated for rules

Tip: Always when in trouble, and you suspect access vector cache denial, use `audit2allow`. If you can't figure out what is going on, also check out the output of `audit2why`, similar tool that produces more human readable reasons why access was denied. Beware though, `audit2why` is somewhat heavy.

Example type enforcement rules

SELinux rules are actually quite simple. For instance the following rule tells to *allow* the process that has context `plonectl_exec_t` access to most common temporary files (`tmp_t`, defined in the reference policy), and the level of access will allow it most of the things that are usually done to files (but not all, for instance `setattr` is missing):

```
allow plonectl_exec_t tmp_t:file { write execute read create unlink open getattr };
```

For the previous to be usable the `tmp_t` and `file` have to be introduced to the compiler, that will search for them from the other available policies. Type is a grouping item that will usually point to a security context (labeled files), while classes define what access types (ie. `getattr`) can be available for the type. The term *type enforcement rule* comes from the fact that SELinux rules define who can do what to the objects that are linked to types.

```
requires {
    type tmp_t { write execute read create unlink open getattr };
    class file;
}
```

There are also macros that will help in accomplishing more complex tasks. The following macro will give the executable right to bind to 8080/TCP:

```
corenet_tcp_bind_transproxy_port(plonectl_exec_t)
```

To get an idea about what items are available the [Reference policy API documentation](#) is the place to go to.

Caveats

First of all, `audit2allow` is not a silver bullet. There are cases where your application accesses something that it does not really require for operation, for instance to scan your system for automatic configuration of services. There are also cases where it prints nothing yet the application clearly is denied access to something. That can be caused by *dontaudit* rules, which silence logging of events that could generate too much noise. In any case a healthy amount of criticism should be applied to everything `audit2allow` output, especially when the suggested rules would give access rights to outside application directories.

Misconfiguration can cause either file labeling to fail, or the application process not to get transitioned to proper executing context. If it seems that the policy is doing nothing, check that the files are labeled correctly (`ls -lFZ`), and the process is running in the correct context (`ps -efZ`).

Evaluating the file context rules (fules and their labels) is managed by a heuristic algorithm, which gives precedence to more specific rules by evaluating the length and precision of the path patterns. The patterns are easy for beginner to misconfigure. When suspecting that the file context rules are not getting applied correctly, always investigate `semanage fcontext -l` to see what rules match your files.

Policies for Plone

The following contains results of ordinary “install, test & break, add rules, repeat from beginning” development cycle for a basic Plone SELinux policy.

Relabeling rights

By default you might not have the right to give any of new security labels to files, and *restorecon* may throw permission denied errors. To give the SELinux utilities (using the context *setfiles_t*) the right to change the security context based on the new types add the following rules:

```
require {
    type setfiles_t;
    type fs_t;
    class lnk_file relabelto;
    class dir relabelto;
    class lnk_file relabelto;
}

allow plone_t fs_t:filesystem associate;
allow setfiles_t plone_t:dir relabelto;
allow setfiles_t plone_t:file relabelto;
allow setfiles_t plone_t:lnk_file relabelto;
allow setfiles_t plonectl_exec_t:dir relabelto;
allow setfiles_t plonectl_exec_t:file relabelto;
allow setfiles_t plonectl_exec_t:lnk_file relabelto;
allow setfiles_t plone_rw_t:dir relabelto;
allow setfiles_t plone_rw_t:file relabelto;
allow setfiles_t plone_rw_t:lnk_file relabelto;
# Python interpreter creates pyc files, this is required to relabel them correctly in
↳ some cases
allow setfiles_t plone_t:file relabelfrom;
```

If the transition is not done, the application will keep running in the starting user's original context. Most likely that will be `unconfined_t`, which means no SELinux restrictions will be applied to the process.

Transition to context

When you first run Plone (i.e. “`plonectl fg`”), you will notice that it doesn’t run, complaining about bad interpreter. Audit2allow will instruct to give rights to your `uncontained_t` context to run the python interpreter. This is however wrong. You wish to first instruct SELinux to change the process always to the new context (`plonectl_exec_t`) when the application is run. You also wish to have the necessary rights to execute the application so that the context transition can start:

```
require {
    type unconfined_t;
    class process { transition siginh noatsecure rlimitinh };
}
# unconfined_r user roles have access to plonectl_exec_t
role unconfined_r types plonectl_exec_t;
# unconfined process contexts should also have execution rights to the python_
↳ executable etc
allow unconfined_t plone_t:file execute;
```

```
# When unconfined_t runs something that has plonectl_exec_t transition the execution
↳context to it
type_transition unconfined_t plonectl_exec_t:process plonectl_exec_t;
# Allow the previous, and some basic process control
allow unconfined_t plonectl_exec_t:process { siginh rlimitinh noatsecure transition };
# The new process probably should have rights to itself
allow plonectl_exec_t self:file entrypoint;
```

Later when enough rules are in place for the application to run take a look at the process context to see that the transitioning to *plonectl_exec_t* works:

```
# ps -efZ|grep python
unconfined_u:unconfined_r:plonectl_exec_t:s0-s0:c0.c1023 root 1782 1 0 16:32 ?_
↳00:00:00 /usr/local/Plone/Python-2.7/bin/python ...
unconfined_u:unconfined_r:plonectl_exec_t:s0-s0:c0.c1023 500 1784 1782 8 16:32 ?_
↳00:00:07 /usr/local/Plone/Python-2.7/bin/python ...
```

Common process requirements

In order for any *NIX process to work some basic requirements must be met. Applications require for instance access to /dev/null, and PTYs:

```
dev_rw_null(plonectl_exec_t)
domain_type(plonectl_exec_t)
files_list_root(plonectl_exec_t)
unconfined_sigchld(plonectl_exec_t)
dev_read_urand(plonectl_exec_t)
userdom_use_inherited_user_ptys(plonectl_exec_t)
miscfiles_read_localization(plonectl_exec_t)
```

Zope/PLONE

After running the plonectl commands (fg, start, stop) several times, and adding the required rules you should end up with something like following. First you will have a large amount of require stanzas for the rule compiler, and then an intermediate amount of rules:

```
require {
    class dir { search read create write getattr rmdir remove_name open add_name };
    class file { rename setattr read lock create write getattr open append };
    type tmp_t;
}

# Read access to common Plone files
allow plonectl_exec_t plone_t:dir { search read open getattr add_name };
allow plonectl_exec_t plone_t:file { execute read create getattr execute_no_trans_
↳ioctl open };
allow plonectl_exec_t plone_t:lnk_file { read getattr };

# Read/write access rights to var and temporary files
allow plonectl_exec_t plone_rw_t:dir { search unlink read create write getattr rmdir_
↳remove_name open add_name };
allow plonectl_exec_t plone_rw_t:file { unlink rename setattr read lock create write_
↳getattr open append };
allow plonectl_exec_t tmp_t:file { unlink rename execute setattr read create write_
↳getattr unlink open };
```

```
allow plonectl_exec_t tmp_t:dir add_name;
fs_search_tmpfs(plonectl_exec_t)
fs_manage_tmpfs_dirs(plonectl_exec_t)
fs_manage_tmpfs_files(plonectl_exec_t)
allow plonectl_exec_t tmpfs_t:file execute;
files_delete_tmp_dir_entry(plonectl_exec_t)

# Networking capabilities
allow plonectl_exec_t self:netlink_route_socket { write getattr read bind create_
↳nlmsg_read };
allow plonectl_exec_t self:tcp_socket { setopt read bind create accept write getattr_
↳getopt listen };
allow plonectl_exec_t self:udp_socket { write read create ioctl connect };
allow plonectl_exec_t self:unix_stream_socket { create connect };
corenet_tcp_bind_generic_node(plonectl_exec_t)
corenet_tcp_bind_http_cache_port(plonectl_exec_t)

# Ability to fork to background, and to communicate with child processes via socket
allow plonectl_exec_t self:process { fork sigchld };
allow plonectl_exec_t plone_rw_t:sock_file { create link write read unlink setattr };
allow plonectl_exec_t self:unix_stream_socket connectto;
allow plonectl_exec_t self:capability { setuid setgid };

# Rights to managing own process
allow plonectl_exec_t self:capability { kill dac_read_search dac_override };
allow plonectl_exec_t self:process { signal sigkill };
```

Gathering the previous audit2allow failed completely to report tcp_socket read and write. Some system policy had probably introduced a *dontaudit* rule, which quiesced the logging for that access vector denial. Luckily Plone threw out very distinct Exception, which made resolving the issue easy.

ZEO

There are couple differences between standalone and ZEO installations. To support both a boolean is probably good way to go. Booleans can be managed like:

```
# getsebool ploneZEO
ploneZEO --> off
# setsebool ploneZEO=true
# setsebool ploneZEO=false
```

Installing Plone in ZEO mode will change the directory *zinstance* to *zeocluster*. It is alright to either have both defined in **plone.fc**, or to use regexp:

```
/usr/local/Plone/zeocluster/var.* gen_context(system_u:object_r:plone_rw_t,s0)
# or
/usr/local/Plone/(zinstance|zeocluster)/var.* gen_context(system_u:object_r:plone_rw_
↳t,s0)
```

The differences to type enforcement policy consist mostly of more networking abilities (which one probably should not allow unless really required), and the ability to run shells (ie. bash):

```
require {
    type bin_t;
    type shell_exec_t;
}
```



```
# ZEO
bool ploneZEO false;
if (ploneZEO) {
allow plonectl_exec_t plone_t:file execute_no_trans;
allow plonectl_exec_t self:tcp_socket connect;
corenet_tcp_bind_transproxy_port(plonectl_exec_t)
nis_use_yppbind_uncond(plonectl_exec_t)
# Starting ZEO requires running shells
kernel_read_system_state(plonectl_exec_t)
allow plonectl_exec_t shell_exec_t:file { read open execute };
}
```

Maintenance utilities

The procedure for allowing maintenance utilities like *buildout* to work is quite straight forward. First introduce a new context:

```
type plone_maint_exec_t;
files_type(plone_maint_exec_t)
```

Then label the maintenance utilities using the context:

```
/usr/local/Plone/zinstance/bin/buildout gen_context(system_u:object_r:plone_maint_
↪exec_t,s0)
```

Last, provide the necessary rules for relabeling, context transition, and for the process to run without any restrictions:

```
role unconfined_r types plone_maint_exec_t;
allow unconfined_t plone_maint_exec_t:file execute;
type_transition unconfined_t plone_maint_exec_t:process plone_maint_exec_t;
allow unconfined_t plone_maint_exec_t:process { siginh rlimitinh noatsecure_
↪transition };
allow plone_maint_exec_t self:file entrypoint;

# Allow anything labeled plone_maint_exec_t to do basically anything
permissive plone_maint_exec_t;
```

After running maintenance tasks you should make sure the files have still correct labels by running something like:

```
/sbin/restorecon -F -R /usr/local/Plone
```

Tip: See also “setenforce Permissive”, which will disable enforcing SELinux rules temporarily system wide.

Testing the policy

Easiest way to test the policy is to label for instance the Python executable as `plone_exec_t` by using *chcon*, and to test the policy using Python scripts. For example:

```
# cd /usr/local/Plone/Python2.7/bin
# setenforce Permissive
# chcon system_u:object_r:plonectl_exec_t:s0 python2.7
```

```
# setenforce Enforcing
# ./python2.7
Python 2.7.3 (default, Apr 28 2013, 22:22:46)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-3)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import os
>>> os.listdir('/root')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OSError: [Errno 13] Permission denied: '/root'
>>> # That should have worked, running python interpreter as root and all
>>> exit()
# setenforce Permissive
# chcon system_u:object_r:plonectl_t:s0 python2.7
# setenforce Enforcing
```

This can be refined into automated testing. Other forms such as Portlet inside running Plone process can also be used for testing.

Deploying the policy

SELinux policies can be installed simply by running `semodule -n -i <compiled_policy.pp>`. In case packaging is required (for rolling out Plone instances automatically, or for use with centralized management tools like Satellite) it is easy to accomplish with rpm. In order to do that first install the rpm building tools:

```
yum install rpm-build
```

Then modify the following RPM spec file to suit your needs:

```
%define relabel_files() \
restorecon -R /usr/local/Plone; \

%define selinux_policyver 3.7.19-195

Name:    plone_selinux
Version: 1.0
Release: 1%{?dist}
Summary: SELinux policy module for plone

Group:   System Environment/Base
License: GPLv2+
# This is an example. You will need to change it.
URL:     http://setest
Source0: plone.pp
Source1: plone.if

Requires: policycoreutils, libselinux-utils
Requires(post): selinux-policy >= %{selinux_policyver}, policycoreutils
Requires(postun): policycoreutils
Requires(post): python
BuildArch: noarch

%description
This package installs and sets up the SELinux policy security module for plone.

%install
```

```

install -d %{buildroot}%{_datadir}/selinux/packages
install -m 644 %{SOURCE0} %{buildroot}%{_datadir}/selinux/packages
install -d %{buildroot}%{_datadir}/selinux/devel/include/contrib
install -m 644 %{SOURCE1} %{buildroot}%{_datadir}/selinux/devel/include/contrib/

%post
semodule -n -i %{_datadir}/selinux/packages/plone.pp
if /usr/sbin/selinuxenabled ; then
    /usr/sbin/load_policy
    %relabel_files
fi;
exit 0

%postun
if [ $1 -eq 0 ]; then
    semodule -n -r plone
    if /usr/sbin/selinuxenabled ; then
        /usr/sbin/load_policy
        %relabel_files
    fi;
fi;
exit 0

%files
%attr(0600,root,root) %{_datadir}/selinux/packages/plone.pp
%{_datadir}/selinux/devel/include/contrib/plone.if

%changelog
* Wed May 1 2013 YOUR NAME <YOUR@EMAILADDRESS> 1.0-1
- Initial version

```

The rpm packages will be built by running the rpmbuild:

```

# rpmbuild -ba plone.spec
# ls -lF /root/rpmbuild/RPMS/noarch/
-rw-r--r--. 1 root root 17240 1.5. 19:24 plone_selinux-1.0-1.el6.noarch.rpm

```

External resources

The following external resources are sorted by probable usefulness to someone who is beginning working with SELinux:

- [Fedora SELinux FAQ](#)
- [Reference policy API](#)
- [NSA - SELinux FAQ](#)
- [NSA - SELinux main website](#)
- [Official SELinux project wiki](#)
- [Red Hat Enterprise SELinux Policy Administration \(RHS429\) classroom course](#)
- [Tresys Open Source projects](#) (IDE, documentation about the reference policy, and several management tools)

Sessions and cookies

Sessions

Description

How Plone handles anonymous and logged-in user sessions. How to store and retrieve session data variables programmatically.

Introduction

Sessions are visitor sessions at the site.

Sessions have features like:

- Login and logout, but also identified by a cookie
- Timeout
- Hold arbitrary per-user data on server side
- Identified by cookies

In Plone, sessions are managed by Zope's `session_data_manager` tool. The source code is in [Products.Sessions](#).

Setting a session parameter

Plone has a tool called `session_data_manager`.

Example:

```
sdm = self.context.session_data_manager
session = sdm.getSessionData(create=True)
session.set("my_option", any_python_object_supporting_pickling)
```

Getting a session

Plone has a convenience method to get the session of the current user:

```
session = sdm.getSessionData(create=True)
```

Getting session id

Each session has a unique id associated with it, for both both anonymous and logged-in users.

Session data is stored in browser cookies, so sessions are browser-specific. If the same user has multiple browsers open on your site, each browser will have its own session.

If you need to refer to the session id, you can query for it:

```
sdm = self.context.session_data_manager
session_id = sdm.getBrowserIdManager().getBrowserId(create=False)
# Session id will be None if the session has not been created yet
```

Initial construction of session data

The example below creates a session data variable when it is accessed for the first time. For the subsequent accesses, the same object is returned. The object changes are automatically persisted if it inherits from the `persistent.Persistent` class.

Note: Session data stored this way does not survive Plone restart.

Example:

```
def getOrCreateCheckoutSession(context, create=False, browser_id=None):
    """ Get the named session object for storing session data.

    Each add-on product can have their own session data slot(s)
    identified by a string name.

    @param context: Any Plone content item with acquisition support

    @param create: Force new data creation, otherwise return None if not exist

    @param browser_id: Cookie id in the user browsers. We can set this
        explicitly if we want to

    @return: ICheckoutData instance
    """

    session_manager = context.session_data_manager
    if browser_id is None:
        if not session_manager.hasSessionData() and not create:
            return
        session = session_manager.getSessionData()
    else:
        session = session_manager.getSessionDataByKey(browser_id)
        if session is None:
            return
    if not session.has_key(CHECKOUT_DATA_SESSION_KEY):
        if create:
            session[CHECKOUT_DATA_SESSION_KEY] = CheckoutData()
        else:
            return None
```

Deleting session data

Example:

```
def _destroyCartForSession(self, context, browser_id=None):
    session_manager = getToolByName(context, 'session_data_manager')
    if browser_id is None:
        if not session_manager.hasSessionData(): #nothing to destroy
            return None
        session = session_manager.getSessionData()
    else:
        session = session_manager.getSessionDataByKey(browser_id)
        if session is None:
            return
```

```
if not session.has_key('getpaid.cart') :  
    return  
del session['getpaid.cart']
```

Session data and unit testing

- Please see <http://article.gmane.org/gmane.comp.web.zope.plone.user/104243>

Using Plone authentication cookie in other systems

- http://stackoverflow.com/questions/12167202/how-to-wrap-plone-authentication-around-a-third-party-servlet/12171528#comment16307483_12171528

Exploring Plone session configuration

- <http://stackoverflow.com/questions/12211682/how-to-export-plone-session-configuration>

Cookies

Description

Handling session and other cookies in Plone

Introduction

Setting and getting cookies

- <http://www.dieter.handshake.de/pyprojects/zope/book/chap3.html>
- <http://stackoverflow.com/questions/1034252/how-do-you-get-and-set-cookies-in-zope-and-plone>

Reading cookies

Usually you want to read incoming cookies sent by the browser.

Example:

```
self.request.cookies.get("cookie_name", "default_value_if_cookie_not_set")
```

Setting cookies

See `HTTPResponse.setCookie()`.

Modifying HTTP response cookies

You might want to tune up or clean cookies after some other part of Plone code has set them. You can do this in *post-publication event handler*.

Example `cleancookies.py` (needs ZCML subscriber registration too):

```
"""
    Clean I18N cookies from non-HTML responses so that e.g. Image
    content, which has language set, and is cross-linked across page,
    don't inadvertently change the language.
"""

from zope.interface import Interface
from zope.component import adapter
from plone.postpublicationhook.interfaces import IAfterPublicationEvent

@adapter(Interface, IAfterPublicationEvent)
def clean_language(object, event):
    """ Clean up cookies after HTTPResponse object has been constructed completely.

    Post-publication handler.
    """
    request = event.request

    #print "%s %s" % (request["URL"], request.response.cookies)

    # All non-HTML payloads
    if not request.response.headers["content-type"].startswith("text/html"):
        # Rip-off I18N_language cookie
        if "I18N_LANGUAGE" in request.response.cookies:
            print "Cleaned up cookie for %s" % request["URL"]
            del request.response.cookies["I18N_LANGUAGE"]
```

Default Plone cookies

Typical Plone cookies:

```
# Logged in cookie
__ac="NjE2NDZkNjk2ZTMwOjcyNzQ3NjQxNjQ2ZDY5NmUzNjM2MzczNw%253D%253D";

# Language chooser
I18N_LANGUAGE="fi";

# Status message
statusmessages="BURUZXJ2ZXR1bG9hISBPbGV0IG55dCBraXJqYXV0dW51dCBzaXPDpMOKbi5pbmZv"

# Google Analytics tracking
__utma=39444192.1440286234.1270737994.1321356818.1321432528.21;
__utmz=39444192.1306272121.6.1.utmcsr=(direct)|utmccn=(direct)|utmcmd=(none);
__utmb=39444192.3.10.1321432528;
__utmc=39444192;
```

```
# Plone copy-paste clipboard
__cp="x%25DA%2515%258AA%250A%25800%250C%2504%25A3%25A0%25E0E%257CF%25FF%25E4%2529%2587
↪%25801%25D5B%25B3-%25F8%257B%25D3%25C3%250E%25CC%25B0i%2526%2522%258D%25D19%2505
↪%25D2%2512%25C0P%25DF%2502%259D%25AB%253E%250C%2514_%25C3%25CAu%258B%25C0%258Fq
↪%2511s%25E8k%25EC%250AH%25FE%257C%258Fh%25AD%25B3qm.9%252B%257E%25FD%25D1%2516%25B3
↪"; Path=/
```

Zope session cookie

This cookie looks like:

```
_ZopeId="25982744A40dimYreFU"
```

It is set first time when session data is written.

Language cookie

I18N_LANGUAGE is set by portal_languages tool. Disable it by *Use cookie for manual override* setting in portal_languages.

Also, language cookie has a special lifecycle when plone.app.multilingual is installed. This may affect your front-end web server caching. If configured improperly, the language cookie gets set on images and static assets like CSS HTTP responses.

- <http://stackoverflow.com/questions/5715216/why-plone-3-sets-language-cookie-to-css-js-registry-files-and-how-to-get-rid-o>

Session cookie lifetime

Setting session cookie lifetime

- <https://plone.org/documentation/kb/cookie-duration>

Sanitizing cookies for the cache

You don't want to store HTTP responses with cookies in a front end cache server, because this would be a leak of other users' information.

Don't cache pages with cookies set. Also with multilingual sites it makes sense to have unique URLs for different translations as this greatly simplifies caching (you can ignore language cookie).

Note that cookies can be set:

- by the server (Plone itself)
- on the client side, by JavaScript (Google Analytics)

... so you might need to clean cookies for both incoming HTTP requests and HTTP responses.

More info in Varnish section of this manual.

Signing cookies

Kind of... crude example

- <https://gist.github.com/3951630>

Status messages

Status messages are session-bound information which allow the user to see notifications when the page is rendered next time.

Status messages are stored session in safely manner which prevents Cross-Site Scripting attacks which might occur due to delivering message information as HTTP GET query parameters.

Setting a status message

Status messages have text (unicode) and type (str). All pending status messages are shown to the user when the next page is rendered.

Example:

```
from Products.statusmessages.interfaces import IStatusMessage

messages = IStatusMessage(self.request)

messages.add(u"Item deleted", type=u"info")
```

Example which you can use in Python scripts:

```
# This message is in Plone i18n domain
context.plone_utils.addPortalMessage(_(u'You are now logged in. Welcome to supa-dupa-
↪system.'), 'info')
```

Rendering status message style by hand-crafted HTML

If you want to insert elements looking status messages on your page use the following mark-up

```
<dl class="portalMessage error">
  <dt>Error</dt>
  <dd>Login failed. Both login name and password are case sensitive, check that_
↪caps lock is not enabled.</dd>
</dl>
```

Login and logout

Description

Login and logout related programming activities in Plone

Introduction

This chapter contains login and logout related code snippets.

Login entry points

There are two login points in Plone

`/login` view (appended to any content URL) directs you to the page where you came from after the login.

`/login_form` view does login without the redirect back to the original page.

In addition, the `/logout` action logs the user out.

The logic that drives the login process is implemented using the CMF form controller framework (legacy). To customize it, you need to override one or more of the `login_*` scripts. This can be accomplished in two ways: register your own skin directory or use `z3c.jbot`. Note that in both cases, you need to copy the `.metadata` file as well.

Extracting credentials

Extracting credentials try to extract log-in (username, password) from HTTP request.

Below is an example how to extract and authenticate the user manually. It is mostly suitable for unit testing. Note that given login field isn't necessarily the username. For example, `betahaus.emaillogin` add-on authenticates users by their email addresses.

Credential extraction will go through all plug-ins registered for `PlonePAS` system.

The first found login/password pair attempt will be used for user authentication.

Unit test example:

```
def extract_credentials(self, login, password):
    """ Spoof HTTP login attempt.

    Functional test using zope.testbrowser would be
    more appropriate way to test this.
    """

    request = self.portal.REQUEST

    # Assume publishing process has succeeded and object has been found by traversing
    # (this is usually set by ZPublisher)
    request['PUBLISHED'] = self.portal

    # More ugly ZPublisher stubs
    request['PARENTS'] = [self.portal]
    request.steps = [self.portal]

    # Spoof HTTP request login fields
    request['__ac_name'] = login
    request['__ac_password'] = password

    # Call PluggableAuthService._extractUserIds()
    # which will return a list of user ids extracted from the request
    plugins = self.portal.acl_users.plugins

    users = self.portal.acl_users._extractUserIds(request, plugins)
```

```

if len(users) == 0:
    return None

self.assertEqual(len(users), 1)

# User will be none if the authentication fails
# or anonymous if there were no credential fields in HTTP request
return users[0]

```

Authenticating the user

Using username and password

Authenticating the user will check that username and password are correct.

Pluggable Authentication Service (acl_users under site root) will go through all authentication plug-ins and return the first successful authenticated users.

Read more in [PlonePAS](#).

Unit test example:

```

def authenticate_using_credentials(self, login, password):

    request = self.portal.REQUEST

    # Will return valid user object
    user = self.portal.acl_users.authenticate(login, password, request)
    self.assertNotEqual(user, None)

```

Using username only

Useful for sudo style logins.

```

def loginUser(self, username):
    """
    Login Plone user (without password)
    """
    self.context.acl_users.session._setupSession(username, self.context.REQUEST.
↪RESPONSE)
    self.request.RESPONSE.redirect(self.portal_state.portal_url())

```

See also

- https://github.com/miohtama/niteoweb.loginas/blob/master/niteoweb/loginas/browser/login_as.py

Post-login actions

Post-login actions are executed after a successful login. Post-login actions which you could want to change are

- Where to redirect the user after login
- Setting the status message after login

You can use the `collective.onlogin` package to set up many actions for you.

If you need more control, post-login code can be executed with *events* defined in `PluggableAuthService` service.

- `IUserLoggedInEvent`
- `IUserInitialLoginInEvent` (logs for the first time)
- `IUserLoggedOutEvent`

Here is an example how to redirect a user to a custom folder after he/she logs in (overrides standard Plone login behavior)

`configure.zcml`:

```
<configure
  xmlns="http://namespaces.zope.org/zope"
  i18n_domain="my.package">

  <subscriber
    for="Products.PluggableAuthService.interfaces.events.IUserLoggedInEvent"
    handler=".postlogin.logged_in_handler"
  />

</configure>
```

`postlogin.py`:

```
# Python imports
import logging

# ZODB imports
from ZODB.POSException import ConflictError

# Zope imports
from AccessControl import getSecurityManager
from zope.interface import Interface
from zope.component import getUtility
from zope.app.component.hooks import getSite
from zope.globalrequest import getRequest

# CMFCore imports
from Products.CMFCore import permissions

# Plone imports
from Products.CMFPlone.interfaces.siteroot import IPloneSiteRoot

# Logger output for this module
logger = logging.getLogger(__name__)

#: Site root relative path where we look for the folder with an edit access
CUSTOM_USER_FOLDERS = "fi/yritykset"

def redirect_to_edit_access_folder(user):
    """
    Redirects the user to a folder he/she has editor access.

    This is for use cases where you have a owned content
    (e.g. company/product data) on a shared site.
```

```

    You want to make it simple for the users with limited knowledge to edit their own
    ↪data
    by redirecting to the edit view right after the login.

    :return: URL if we should redirect, otherwise None
    """

    # Fetch the site related to the current HTTP request
    portal = getSite()

    # Look for portal relative paths where the items are
    try:
        target = portal.unrestrictedTraverse(CUSTOM_USER_FOLDERS)
    except ConflictError:
        # Transaction retries must be
        # always handled specially in exception handlers
        raise
    except Exception, e:
        # Let the login proceed even if the folder has been deleted
        # don't make it impossible to login to the site
        logger.exception(e)
        return None

    # Check if the current user has Editor access
    # in the any items of the folder
    sm = getSecurityManager()

    for obj in target.listFolderContents():
        if sm.checkPermission(permissions.ModifyPortalContent, obj):
            logger.info("Redirecting user %s to %s" % (user, obj))
            return obj.absolute_url() + "/edit"

    logger.warn("User %s did not have his/her own content item in %s" % (user,
    ↪target))

    # Let the normal login proceed to the page "You are now logged in" etc.
    return None

def logged_in_handler(event):
    """
    Listen to the event and perform the action accordingly.
    """

    user = event.object

    url = redirect_to_edit_access_folder(user)
    if url:
        request = getRequest()
        if request is None:
            # HTTP request is not present e.g.
            # when doing unit testing / calling scripts from command line
            return

        # check if came_from is not empty, then clear it up, otherwise further
        # Plone scripts will override our redirect
        if request.get('came_from', None):
            request['came_from'] = ''

```

```
request.form['came_from'] = ''
request.RESPONSE.redirect(url)
```

Post-logout actions

Products.PlonePAS.tools.membership fires `Products.PlonePAS.events.UserLoggedOutEvent` when the user logs out via *Log out* menu item.

Note: You cannot catch session timeout events this way... only explicit logout action.

Example ZCML

```
<subscriber for="Products.PlonePAS.events.UserLoggedOutEvent"
  handler=".smartcard.clear_extra_cookies_on_logout" />
```

Example Python:

```
def clear_extra_cookies_on_logout(event):
    """
    Logout event handler.

    When user explicitly logs out from the Logout menu, clear our privileges_
    ↪ smartcard cookie.
    """

    # Which cookie we want to clear
    cookie_name = SmartcardHelper.PRIVILEGED_COOKIE_NAME

    request = event.object.REQUEST
    # YES CAPS LOCK WAS MUST WHEN ZOPE 2 WAS INVENTED
    # SOMEWHERE AROUND NINETIES. THEN IT WAS THE CRUISE
    # CONTROL FOR COOLNESS AND ZOPE IS SOO COOOOOL.
    response = request.RESPONSE
    # Voiding our special cookie on logout
    response.expireCookie(cookie_name)
```

More info

- <https://github.com/plone/Products.PlonePAS/blob/master/Products/PlonePAS/tools/membership.py#L645>

Entry points to logged in member handling

See `Products.PluggableAuthService.PluggableAuthService._extractUserIds()`. It will try to extract credentials from incoming HTTP request, using different “extract” plug-ins of PAS framework.

`PluggableAuthService` is also known as `acl_users` persistent object in the site root.

For each set of extracted credentials, try to authenticate a user; accumulate a list of the IDs of such users over all our authentication and extraction plugins.

`PluggableAuthService` may use *ZCacheable* pattern to see if the user data exists already in the cache, based on any extracted credentials, instead of actually checking whether the credentials are valid or not. `PluggableAuthService` must be set to have cache end. By default it is not set, but installing LDAP sets it to RAM cache.

More info

- <https://github.com/plone/plone.app.ldap/blob/master/plone/app/ldap/ploneldap/util.py>

PAS cache settings

Here is a short view snippet to set PAS cache state:

```
from Products.Five.browser import BrowserView
from zope.app.component.hooks import getSite

from Products.CMFCore.utils import getToolByName

class PASCacheController(BrowserView):
    """
    Set PAS caching parameters from browser address bar.
    """

    def getPAS(self):
        site=getSite()
        return getToolByName(site, "acl_users")

    def setPASCache(self, value):
        """
        Enables or disables pluggable authentication service caching.

        The setting is stored persistently in PAS

        This caches credentials for authenticated users after the first login.

        This will make authentication and permission operations little bit faster.
        The downside is that the cache must be purged if you want to remove old_
        ↪values from there.
        (user has been deleted, etc.)

        More info

        * https://github.com/plone/plone.app.ldap/blob/master/plone/app/ldap/
        ↪ploneldap/util.py

        """

        pas = self.getPAS()

        if value:

            # Enable

            if pas.ZCacheable_getManager() is None:
                pas.ZCacheable_setManagerId(manager_id="RAMCache")

            pas.ZCacheable_setEnabled(True)

        else:

            # Disable
            pas.ZCacheable_setManagerId(None)
            pas.ZCacheable_setEnabled(False)
```

```
def __call__(self):
    """ Serve HTTP GET queries.
    """

    cache_value = self.request.form.get("cache", None)

    if cache_value is None:
        # Output help text
        return "Use: http://localhost/@@pas-cache-controller?cache=true"

    value = (cache_value == "true")

    self.setPASCACHE(value)

    return "Set value to:" + str(value)
```

... and related ZCML

```
<browser:page
  for="Products.CMFCore.interfaces.ISiteRoot"
  name="pas-cache-controller"
  class=".pascache.PASCACHEController"
  permission="cmf.ManagePortal"
/>
```

Login as another user (“sudo”)

If you need to login to production system another user and you do not know the password, there is an add-on product for it

- <https://pypi.python.org/pypi/niteoweb.loginas>

Another option

- <https://pypi.python.org/pypi/Products.OneTimeTokenPAS>

Getting logged in users

Locking user account after too many retries

For security reasons, you might want to locking users after too many tries of logins. This protects user accounts against brute force attacks.

- <https://github.com/collective/Products.LoginLockout/tree/master/Products/LoginLockout>

Hyperlinks to authenticated Plone content in Microsoft Office

Microsoft Office applications (in the first instance Word and Excel), have been observed to attempt to resolve hyperlinks once clicked, prior to sending the hyperlink to the user’s browser. If such a link points to some Plone content that requires authentication, the Office application will request the URL first, and receive a 302 Redirect to the `require_login` Python script on the relevant Plone instance. If your original hyperlink was like so:

```
http://example.com/myfolder/mycontent
```


and this URL requires authentication, then the Office application will send your browser to this URL:

```
http://example.com/acl_users/credentials_cookie_auth/require_login?came_from=http%3A//
↪example.com/myfolder/mycontent
```

Normally, this isn't a problem if a user is logged out at the time. They will be presented with the relevant login form and upon login, they will be redirected accordingly to the `came_from=` URL.

However, if the user is *already* logged in on the site, visiting this URL will result in an `Insufficient Privileges` page being displayed. This is to be expected of Plone (as this URL is normally only reached if the given user has no access), but because of Microsoft Office's mangling of the URL, may not necessarily be correct as the user may indeed have access.

The following drop-in replacement for the `require_login` script has been tested in Plone 4.1.3 (YMMV). Upon a request coming into this script, it attempts (a hack) to traverse to the given path. If permission is actually allowed, Plone redirects the user back to the content. Otherwise, things proceed normally and the user has no access (and is shown the appropriate message):

```
## Script (Python) "require_login"
##bind container=container
##bind context=context
##bind namespace=
##bind script=script
##bind subpath=traverse_subpath
##parameters=
##title=Login
##

login = 'login'

portal = context.portal_url.getPortalObject()
# if cookie crumbler did a traverse instead of a redirect,
# this would be the way to get the value of came_from
#url = portal.getCurrentUrl()
#context.REQUEST.set('came_from', url)

if context.portal_membership.isAnonymousUser():
    return portal.restrictedTraverse(login)()
else:
    expected_location = context.REQUEST.get('came_from')
    try:
        #XXX Attempt a traverse to the given path
        portal.restrictedTraverse(expected_location.replace(portal.absolute_url()+'/',
↪'))
        container.REQUEST.RESPONSE.redirect(expected_location)
    except:
        return portal.restrictedTraverse('insufficient_privileges')()
```

For further reading see:

- <http://plone.293351.n2.nabble.com/Linking-to-private-page-from-MS-Word-redirect-to-login-form-td5495131.html>
- <http://plone.293351.n2.nabble.com/Problem-with-links-to-files-stored-in-Plone-td3055014.html>
- <http://bytes.com/topic/asp-classic/answers/596062-hyperlinks-microsoft-applications-access-word-excel-etc>
- <https://community.jivesoftware.com/docs/DOC-32157>

Single Sign-On and Active Directory

Plone can be used in a Microsoft Active Directory environment (or standard Kerberos environment) such that users are automatically and transparently authenticated to Plone without requesting credentials from the user.

This is quite an advanced topic and requires some set up on the server, but can be achieved with Plone running on either Unix/Linux or Windows environments.

More details can be found in this presentation from Plone Open Garden 2013:

- <http://www.slideshare.net/hammertoe/plone-and-singlesign-on-active-directory-and-the-holy-grail>
- http://www.youtube.com/watch?v=-FLQxeD5_1M

Preventing duplicate logins from different browsers

- <http://stackoverflow.com/questions/2562385/limit-concurrent-user-logins-in-plone-zope>

Images

Image manipulation related documentation.

Image-like content

Description

How to programmatically manipulate images on your Plone site.

Introduction

Plone supports image content in several forms:

stand-alone content type As stand-alone content type, images will be visible in the sitemap. This is the case for the default Image content type, but you can create custom content types with similar properties.

image field As a field, the image is directly associated with one content object. Use `plone.namedfile.field.NamedBlobImage`:

```
custom_image = NamedBlobImage(  
    title=_('label_customimage', default=u'Custom Image'),  
    description=_('help_customimage', default=u ''),  
    required=False,  
)
```

behavior using an image field The `plone.leadimage` behavior in `plone.app.contenttypes.behaviors.leadimage` provides a field called `leadimage`. Custom behaviors like this can be created too.

Custom image content type

If you want to have your custom content type behave like the stock Plone Image content type:

- Inherit from the content class `plone.app.contenttype.content.Image` and use the xml schema from that class.
- When writing the GenericSetup XML of your type, follow the example of [Image.xml](#).
- Do not set workflow for your type in `profiles/default/workflows.xml`.

```
<?xml version="1.0"?>
<object name="portal_workflow" meta_type="Plone Workflow Tool">

  <bindings>
    <type type_id="YourImageType"/>
  </bindings>
</object>
```

Accessing images

Both - Dexterity and Archetypes - are offering the same traversable `@@images` view. It can be used from page templates and Python code to provide access to the image and different image scales for image fields on content.

The code for image access and scales for Dexterity based content is handled by `plone.namedfile`. Old Archetypes based content image scales is handled by `plone.app.imaging`.

Using direct URLs

If your image field is a primary field, like at the default Image content type, then access works by calling the url without any view:

```
http://yoursite/imagecontent
```

The generic way to access any image on your content is an URL like so:

```
http://yoursite/imagecontent/@@images/FIELDNAME
```

Predefined image scales from the configuration settings are accessed this way:

```
http://yoursite/imagecontent/@@images/FIELDNAME/SCALENAME
```

You might find URLs of custom (on-the-fly) image scales accessed this way (see below):

```
http://yoursite/imagecontent/@@images/FIELDNAME/CUSTOM_SCALE_UID.jpg
```

Examples 1, show the original image from the `plone.leadimage` behavior:

```
http://yoursite/imagecontent/@@images/leadimage
```

Examples 2, show the scale `mini` from the field `custom_image`:

```
http://yoursite/imagecontent/@@images/custom_image/mini
```

Access by creating tags programmatically

In code a lookup of the `images` multi-adapter is needed. It implements the `plone.app.imaging.interfaces.IImageScaling` interface, thus it provides:

`scale(fieldname=None, scalename=None, **parameters)` Retrieve a scale based on the given name or set of parameters. The parameters can be anything supported by *scaleImage* and would usually consist of at least a width & height.

Returns either an object implementing *IImageScale* or *None*

`tag(fieldname=None, scalename=None, **parameters)` Like `scale` but returns a tag for a scale.

`getAvailableSizes(fieldname=None)` returns a dictionary of scale name => (width, height)

`getImageSize(fieldname=None)` returns the original image size, a tuple of (width, height)

`getInfo(fieldname=None, scalename=None, **parameters)` returns metadata for the requested scale from the storage

`images` is in fact a view (a multi-adapter between context and request), we can use `plone.api.content.get_view` for lookup:

```
from plone import api

...

scale_util = api.content.get_view('images', context, request)
tag = scale_util.tag('leadimage', 'mini')
```

Creating Scales

Named scales

In the Plone Control Panel under Image Handling images scales can be defined (and redefined). Those scales are stored in the configuration registry. In a custom `GenericSetup` profile additional scales can be added by adding some lines to `registry.xml` like so:

```
<?xml version="1.0"?>
<registry>
  <records>
    interface="Products.CMFPlone.interfaces.controlpanel.IImagingSchema"
    prefix="plone">
      <value key="allowed_sizes" purge="false">
        <element>custom_4to3 400:300</element>
        <element>custom_3to4 300:400</element>
      </value>
    </records>
  ...
</registry>
```

Scales On-The-Fly

Sometimes scales need to be created on-the-fly. This can be done programmatically only. In order to create scale on the fly the `images` multi-adapter is used.

The methods `scale`, `tag` or `getInfo` can be used to create a scale.

In order to create a custom scale skip the `scalesname` parameter and use `height` and `width` parameters.

Optional choose the `direction` parameter:

up Scaling scales the smallest dimension up to the required size and crops the other dimension if needed.

down Scaling starts by scaling the largest dimension to the required size and crops the other dimension if needed.

thumbnail scales to the requested dimensions without cropping. The resulting image may have a different size than requested. This option requires both width and height to be specified. *keep* is accepted as an alternative spelling for this option, but its use is deprecated.

Example, scale down (crop) to 300x200:

```
from plone import api

...

scale_util = api.content.get_view('images', context, request)
tag = scale_util.tag('leadimage', width=300, height=200, direction=down)
```

Attention: The generated URL is based on a generated UID which points to the current scaled down version of the image. After modification of the content type the scale is not updated, but a new URL to the new scale will be generated. But the generated UID will be reused for the same upload, so one version is scaled only once.

portal_catalog and images

Never index image objects or store them as metadata, as adding image data to the `portal_catalog` brain objects would greatly increase their size and make brain look-up slow.

Instead recreate the path of the image

Or if you have custom scales not available in configuration, index only image paths with this scale information using `getPhysicalPath()`.

Addons

Manual croppings can be chosen by using `plone.app.imagecropping`

Images in page templates

Description

How to link to images in page templates in Plone.

Putting a static image into a page template

Here is an example how to create an `` tag in a `.pt` file:

```
<img tal:attributes="src string:${context/@@plone_portal_state/portal_url}/
  ++resource++plonetheme.mfabrik/close-icon.png" alt="[ X ]"/>
```

Let's break this down:

- We are rendering an `` tag.
- The `src` attribute is dynamically generated using a *TALES* expression.
- We use *string comprehension* to create the `src` attribute. Alternatively we could use e.g. the python: *TALES* expression type and embed one line python of code to generate the attribute value.
- We look up a helper view called *plone_portal_state*. This is a `BrowserView` shipped with Plone. Its purpose is to expose different helper methods to page templates and Python code.
- We call `plone_portal_state.portal_url()` method. This will return the root URL of our site. Note that this is not necessary the domain's top-level URL, as Plone sites can be nested in folders, or served on a path among unrelated web properties.
- We append our Zope 3 resource path to our site root URL (see below). This maps to some static media folder in our add-on files on the disk.
- There we point to `close-icon.png` image file.
- We also add the `alt` attribute of the `` tag normally. It is not dynamically generated.

When the page template is generated, the following snippet could look like, for example:

```

```

... or:

```

```

... depending on the site virtual hosting configuration.

Relative image look-ups

Warning: Never create relative image look-ups without prefixing the image source URL with the site root.

Hardcoded relative image path might seem to work:

```

```

... but this causes a different image *base URL* to be used on every page. The image URLs, from the browser point of view, would be:

```

```

... and then in another folder:

```

```

... which prevents the browser from caching the image.

Registering static media folders in your add-on product

Zope 3 resource directory

The right way to put in a static image is to use a Zope 3 resource directory.

- Create folder `yourcompany.product/yourcompany/product/browser/static`.
- Add the following *ZCML* to `yourcompany.product/yourcompany/product/browser/configure.zcml`.

```
<browser:resourceDirectory
    name="yourcompany.product"
    directory="static"
    layer=".interfaces.IThemeSpecific"
/>
```

This will be picked up at the `++resource++yourcompany.product/ static` media path.

Layer is optional: the static media path is available only when your add-on product is installed if the *layer* is specified.

Also see *Resource folders*

Rendering Image content items

You can refer to `ATImage` object's content data download by adding `/image` to the URL:

```
<img alt="" tal:attributes="src string:${context/getImage/absolute_url}/image" />
```

The magic is done in the `__bobo_traverse__` method of `ATImage` by providing traversable hooks to access image download:

- <https://github.com/plone/Products.ATContentTypes/blob/master/Products/ATContentTypes/content/image.py>

Rendering ImageField

Archetypes's `ImageField` maps its data to the content object at attribute which is the field's name. If you have a field `campaignVideoThumbnail` you can generate an image tag as follows:

```
<img class="thumbnail" tal:attributes="src string:${campaign/absolute_url}/
↪campaignVideoThumbnail" alt="Campaign video" />
```

If you need more complex `` output, create a helper function in your `BrowserView` and use Python code to perform the `ImageField` manipulation.

See `ImageField` for more information:

- <https://github.com/plone/Products.Archetypes/blob/master/Products/Archetypes/Field.py>

tag() method

Note: Using `tag()` is discouraged. Create your image tags manually.

Some content provides a handy `tag()` method to generate `` tags with different image sizes.

`tag()` is available on

- Archetypes ImageField
- ATNewsItem
- ATImage
- FSImage (Zope 2 image object on the file-system)

`tag()` is defined in [OFS.Image](#).

Scaling images

`tag()` supports scaling. Scale sizes are predefined. When an `ATImage` is uploaded, various scaled versions of it are stored in the database.

Displaying a version of the image using the “preview” scale:

```
image.tag(scale="preview", alt="foobar text")
```

This will generate:

```

```

Note: If you are not using the `alt` attribute, you should set it to an empty string: `alt=""`. Otherwise screen readers will read the `src` attribute of the `` tag aloud.

In order to simplify accessing these image scales, use [archetypes.fieldtraverser](#). This package allows you to traverse to the stored image scales while still using `AnnotationStorage` and is a lot simpler to get going (in the author’s humble opinion :).

Default scale names and sizes are defined in `ImageField` declaration for custom `ImageFields`. For `ATImage`, those are in [Products.ATContentTypes.content.image](#).

Lightbox style image pop-ups

Plone comes with [plone.app.jquerytools](#) which offers easy integration for lightbox style image pop-ups.

You can use Plone standard image content type, defining scales using [plone.app.imaging](#) or you can define image fields in your schema.

In the example below we define custom image fields in Archetypes schema.

`contenttype.py`:

```
atapi.ImageField(
    'imageTwo',
    widget=atapi.ImageWidget(
        label=_ (u"Kuva #2"),
    ),
    validators=('isNonEmptyFile'),
    languageIndependent=True,
    sizes={
        'thumb': (90, 90),
        'large': (768, 768),
    },
),
```



```

atapi.ImageField(
    'imageThree',
    widget=atapi.ImageWidget(
        label=_ (u"Kuva #3"),
    ),
    validators=('isNonEmptyFile'),
    languageIndependent=True,
    sizes={
        'thumb': (90, 90),
        'large': (768, 768),
    },
),

```

Related view page template file

```

<div class="product-all-images">

    <img class="product-image-preview" tal:condition="context/getImageTwo"
    ↪tal:attributes="src string:${context/absolute_url}/@@images/imageTwo/thumb" alt="" /
    ↪>

    <img class="product-image-preview" tal:condition="context/getImageThree" alt=""
    ↪tal:attributes="src string:${context/absolute_url}/@@images/imageThree/thumb" />

</div>

```

And then we activate all this in a JavaScript using `prepOverlay()` from `plone.app.jquerytools`

```

/*global window,document*/

(function($) {

    "use strict";

    /**
     * Make images clickable and open a bigger version of the image when clicked
     */
    function prepareProductImagePreviews() {

        // https://pypi.python.org/pypi/plone.app.jquerytools/1.4#examples
        $('.product-image-preview')
        .prepOverlay({
            subtype: 'image',
            urlmatch: 'thumb',
            urlreplace: 'large'
        });
    }

    $(document).ready(function() {
        prepareProductImagePreviews();
    });

})(jQuery);

```

Rotating banners

Simple rotating banners can be done with [jQuery Cycle plug-in \(lite\)](#).

Example TAL code... render list of content items and extract one image from each of them

```
<dd class="cycle">

  <tal:hl repeat="obj view/obj">
    <a tal:attributes="href python:view.getLink(obj); title python:view.
    ↪getAltText(obj) " class="outer-wrapper">
      <img tal:attributes="src python:view.getImageURL(obj) " />
    </a>
  </tal:hl>

</dd>
```

Then use the the following JavaScript to bootstrap the cycling

```
(function($) {

  "use strict";

  function rotateBanners() {
    $(".cycle").cycle();
  }

  $(document).ready(function() {
    rotateBanners();
  });

})(jQuery);
```

You need to have this snippet and `jquery.cycle.light.js` in your `portal_javascripts` registry.

You also may need to set pixel height for `cycle` elements, as they use absolute positioning making the element take otherwise 0 pixel of height.

Pillow (PIL Fork)

Plone uses [Pillow](#), a fork of the Python Imaging Library (PIL) for low-level image manipulation and decoding, particularly for GIF, JPEG and PNG image formats.

Installing Pillow

For instructions how to install Pillow and its dependencies, please read:

<https://pillow.readthedocs.org/en/latest/installation.html>

Note particularly the dependencies on the development packages of `libjpeg` and `libz`. Other image format support may be required for special projects, but `libjpeg` and `libz` are required for GIF, JPEG and PNG support.

Syndication

Introduction

In Plone 4.3, there is a new syndication framework that allows you to customize how content in your site is syndicated.

Customize how a content type is syndicated

In this example, we'll show how to customize how News items are syndicated on your site.

Create adapter

We'll create an adapter that overrides the body text:

```
from Products.CMFPlone.browser.syndication.adapters import BaseItem
from Products.CMFPlone.interfaces.syndication import IFeed
from Products.ATContentTypes.interfaces import IATNewsItem
from zope.component import adapts

class NewsFeedItem(BaseItem):
    adapts(IATNewsItem, IFeed)

    @property
    def body(self):
        return 'Cooked:' + self.context.CookedBody()
```

Register Adapter

Example:

```
<adapter
  factory=".NewsFeedItem"
  for="Products.ATContentTypes.interfaces.IATNewsItem
       Products.CMFPlone.interfaces.syndication.IFeed"
  provides="Products.CMFPlone.interfaces.syndication.IFeedItem" />
```

Dexterity type

If the type you're customizing is a dexterity type then Plone will use the `Products.CMFPlone.browser.syndication.DexterityItem` adapter by default for adopting Dexterity content to syndication. `IFeedItem`. You can override the default adapter by registering your own adapter this way:

```
from zope.component import adapts
from Products.CMFPlone.interfaces.syndication import IFeed
from plone.dexterity.interfaces import IDexterityContent
from Products.CMFPlone.browser.syndication.adapters import DexterityItem

class MyAdapter(DexterityItem):
    adapts(IMyType, IFeed)

    @property
    def link(self):
        return '...some custom url'
```

```
guid = link
```

```
<adapter
  factory=".adapters.MyAdapter"
  for="my.package.mytype.IMyType
    Products.CMFPlone.interfaces.syndication.IFeed"
  provides="Products.CMFPlone.interfaces.syndication.IFeedItem" />
```

Register your Folderish type as syndicable

Just make sure it implements the ISyndicable interface:

```
from Products.CMFPlone.interfaces.syndication import ISyndicable

...
class MyFolderishType(object):
    implements(ISyndicable)
...
```

Create your own feed type

Example of creating your own simple feed type and registering it.

Create your feed template:

```
<?xml version="1.0" ?>
<feed xml:base=""
      xml:lang="en"
      xmlns:i18n="http://xml.zope.org/namespaces/i18n"
      xmlns:tal="http://xml.zope.org/namespaces/tal"
      tal:define="feed view/feed;
                  url feed/link;"
      tal:attributes="xml:base url; xml:lang feed/language"
      i18n:domain="Products.CMFPlone">
  <link rel="self"
        href=""
        tal:attributes="href request/ACTUAL_URL" />
  <title type="html" tal:content="feed/title" />
  <subtitle tal:content="feed/description" />
  <updated tal:content="python:feed.modified.ISO8601()" />
  <link tal:attributes="href url" rel="alternate" type="text/html" />
  <id tal:content="string:urn:syndication:${feed/uid}" />
  <tal:repeat repeat="item feed/items">
    <entry tal:define="published item/published;
                      modified item/modified;">
      <title tal:content="item/title"></title>
      <link rel="alternate" type="text/html" href="" tal:attributes="href item/link" />
    </entry>
  </tal:repeat>
</feed>
```

Register the view in ZCML:

```
<browser:page
    for="Products.CMFPlone.interfaces.syndication.ISyndicable"
    class="Products.CMFPlone.browser.syndication.views.FeedView"
    name="myfeed.xml"
    permission="zope2.View"
    template="myfeed.xml.pt"
/>
```

Finally, register the feed view in the control panel *syndication-settings* in the *Allowed Feed Types* setting. You should be able to append a new feed type like this:

```
myfeed.xml|My Feed Type
```

Now, if the *My Feed Type* is enabled on a syndicable item(you'll probably also need to allow editing syndication settings), you'll be able to append *myfeed.xml* onto the url to use the new syndication.

Creating a json feed type

First, we'll create the json feed view class:

```
from Products.CMFPlone.browser.syndication.views import FeedView
import json

class JSONFeed(FeedView):

    def index(self):
        data = []
        feed = self.feed()
        for item in feed.items:
            data.append({
                'link': item.link,
                'title': item.title,
                'description': item.description
            })
        return json.dumps(data)
```

Then register the adapter with ZCML:

```
<browser:page
    for="Products.CMFPlone.interfaces.syndication.ISyndicable"
    class=".JSONFeed"
    name="json"
    permission="zope2.View"
/>
```

Finally, register the feed view in the control panel *syndication-settings* in the *Allowed Feed Types* setting. You should be able to append a new feed type like this:

```
json|JSON
```

Now, if the *JSON* is enabled on a syndicable item(you'll probably also need to allow editing syndication settings), you'll be able to append *json* onto the url to use the new syndication.

Available FeedItem properties to override

If you're inheriting `Products.CMFPlone.browser.syndication.adapters.BaseItem` or `Products.CMFPlone.browser.syndication.adapters.DexterityItem` in an attempt to override the default feed item behavior, these are the properties available to you to override:

- `link`
- `title`
- `description`
- `categories`
- `published`
- `modified`
- `uid`
- `rights`
- `publisher`
- `author`
- `author_name`
- `author_email`
- `body`
- `guid`
- `has_enclosure`
- `file`
- `file_url`
- `file_length`
- `file_type`

Available feed properties to override

If you're inheriting from `Products.CMFPlone.browser.syndication.adapters.FolderFeed` in an attempt to override the functionality of a feed folder or collection, these are the available properties to override:

- `link`
- `title`
- `description`
- `categories`
- `published`
- `modified`
- `uid`
- `rights`
- `publisher`
- `logo`

- icon
- items
- limit
- language

Miscellaneous information

This section describes functions and APIs which are not directly related to any bigger subsystems. Also some other information that does not fit in any specific category

Managing member profile (portal_membership under site root)

Helper views and tools

Introduction

This document explains how to access view and context utilities in Plone.

IPortalState and IContextState

IPortalState defines IContextState view-like interfaces to access miscellaneous information useful for the rendering of the current page.

The views are cached properly, they should access the information effectively.

- IPortalState is mapped as the plone_portal_state name view.
- IContextState is mapped as the plone_context_state named view.
- ITools is mapped as the plone_tools named view.

To see what's available through the interface, read the documentation in the `plone.app.layout.globals.interfaces` module.

Example showing how to get the portal root URL:

```
from zope.component import getMultiAdapter
...

class MyView(BrowserView):
    ...

    def __call__(self):
        # aq_inner is needed in some cases like in the portlet renderers
        # where the context itself is a portlet renderer and it's not on the
        # acquisition chain leading to the portal root.
        # If you are unsure what this means always use context.aq_inner
        context = self.context.aq_inner
        portal_state = getMultiAdapter((context, self.request), name=u'plone_portal_
↪state')

        self.some_url = portal_state.portal_url() + "/my_foo_bar"
```

Example showing how to get the current language:

```
from zope.component import getMultiAdapter

...

portal_state = getMultiAdapter((self.context, self.request), name=u'plone_portal_state
↪')
current_language = portal_state.language()
```

Example showing how to expose portal_state helper to a template:

1. ZCML includes portal_state in allowed_attributes

```
<browser:page
    for="*"
    name="test"
    permission="zope2.Public"
    class=".views.MyView"
    allowed_attributes="portal_state"
/>
```

A Python class exposes the variable:

```
from Acquisition import aq_inner
from zope.component import getMultiAdapter

class MyView(BrowserView):

    def portal_state(self):
        context = aq_inner(self.context)
        portal_state = getMultiAdapter((context, self.request), name=u'plone_portal_
↪state')
        return portal_state
```

Templates can use it as follows:

```
<div>
    The language is <span tal:content="view/portal_state/language" />
</div>
```

You can directly look up portal_state in templates using acquisition and view traversal, without need of ZCML code or Python view code changes. This is useful e.g. in overridden viewlet templates:

```
<!--

    During traversal, ``@@`` signals that the traversing
    machinery should look up a view by that name.

    First we look up the view and then use
    it to access the variables defined in
    ``IPortalState`` interface.

-->

<div tal:define="portal_state context/@@plone_portal_state" >
    The language is <span tal:content="portal_state/language" />
</div>
```


Use in templates and expressions

You can use `IContextState` and `IPortalState` in *TALES* expressions, e.g. `portal_actions`, as well.

Example `portal_actions` conditional expression:

```
python:object.restrictedTraverse('@@plone_portal_state').language() == 'fi'
```

Tools

Tools are persistent utility classes available in the site root. They are visible in the Management Interface, and sometimes expose useful information or configuration here. Tools include e.g.:

portal_catalog Search and indexing facilities for content

portal_workflow Look up workflow status, and do workflow-related actions.

portal_membership User registration information.

Warning: Portal tools are deprecated and are phased out and being replaced by utilities. The [Removal of selected portal tools](#) PLIP is created to migrate from tools to utilities.

Get a portal tool using `plone.api`

It is recommended to use `plone.api` to get a portal tool:

```
from plone import api
catalog = api.portal.get_tool(name='portal_catalog')
```

The `plone.api` package exposes functionality from portal tools, it is not longer necessary to directly call a tool. For example; the API can be used to get the workflow state, change the workflow state, get a member and get the member properties.

ITools interface

`plone.app.layout.globals.interfaces.ITools` interface and `Tools BrowserView` provide cached access for the most commonly needed tools.

`ITools` is mapped as the `plone_tools` view for traversing.

Example:

```
from Acquisition import aq_inner
from zope.component import getMultiAdapter

context = aq_inner(self.context)
tools = getMultiAdapter((context, self.request), name=u'plone_tools')

portal_url = tools.url()

# The root URL of the site is got by using portal_url.__call__()
# method
```

```
the_current_root_url_of_the_site = portal_url()
```

IPlone

`Products.CMFPlone.browser.interfaces.IPlone` provides some helper methods for Plone specific functionality and user interface.

- `IPlone` helper views is registered under the name `plone`

getToolByName

`getToolByName` is the old-fashioned way of getting tools, using the context object as a starting point. It also works for tools which do not implement the `ITools` interface.

`getToolByName` gets any Plone portal root item using acquisition.

Example:

```
from Products.CMFCore.WorkflowCore import WorkflowException

# Do the workflow transition "submit" for the current context
workflowTool = getToolByName(self.context, "portal_workflow")
workflowTool.doActionFor(self.context, "submit")
```

Zope DateTime

Description

Using Zope `DateTime` class in Plone programming

Introduction

Some Plone dates are stored as Zope `DateTime` objects. This is different from standard Python `datetime` (notice the letter casing). Zope `DateTime` predates Python `datetime` which was added in Python 2.4. Zope `DateTime` is old code, so do rites necessary for your religion before programming with it.

- [Zope `DateTime` HTML API documentation](#)
- [Python `datetime` documentation](#)

Note: Using Python `datetime` is recommended if possible. Zope `DateTime` should be dealt in legacy systems only as Python `datetime` is much more documented and widely used.

Default formatting

Since Plone 4

- A per-language format string from a translations is preferred
- If such string is not available the default is taken from portal_properties / site_properties

Formatting examples

US example:

```
localTimeFormat: %b %d, %Y
localLongTimeFormat: %b %d, %Y %I:%M %p
```

European style format:

```
localTimeFormat: %d.%m.%Y (like 1.12.2010) localLongTimeFormat: %H:%M %d.%m.%Y (like 12:59 1.12.2010)
```

More info

- <https://dev.plone.org/wiki/DateTimeFormatting>
- <http://docs.python.org/library/time.html#time.strftime>

DateTime API

Zope DateTime HTML API documentation

You may find the following links useful

- [Source code](#)
- [README](#)
- [Interface description](#)

Converting between DateTime and datetime

Since two different datetime object types are used, you need to often convert between them.

You can convert Zope DateTime objects to datetime objects like so:

```
from DateTime import DateTime
zope_DT = DateTime() # this is now.
python_dt = zope_DT.asdatetime()
```

Vice versa, to convert from a Python datetime object to a Zope DateTime one:

```
zope_DT = DateTime(python_dt)
```

Note, if you use timezone information in python datetime objects, you might loose some information when converting. Zope DateTime handles all timezone information as offsets from GMT.

DateTime problems and pitfalls

This **will fail** silently and you get a wrong date:

```
dt = DateTime("02.07.2010") # Parses like US date 02/07/2010
```

Please see

- <http://pyyou.wordpress.com/2010/01/11/datetime-against-mx-datetime/>

Parsing both US and European dates

Example:

```
# Lazy-ass way to parse both formats
# 2010/12/31
# 31.12.2010
try:
    if "." in rendDate:
        # European
        end = DateTime(rendDate, datefmt='international')
    else:
        # US
        end = DateTime(rendDate)
```

Friendly date/time formatting

Format datetime relative to the current time, human-readable:

```
def format_datetime_friendly_ago(date):
    """ Format date & time using site specific settings.

    @param date: datetime object
    """

    if date == None:
        return ""

    date = DT2dt(date) # zope DateTime -> python datetime

    # How long ago the timestamp is
    # See timedelta doc http://docs.python.org/lib/datetime-timedelta.html
    # since = datetime.datetime.utcnow() - date

    now = datetime.datetime.utcnow()
    now = now.replace(tzinfo=pytz.utc)

    since = now - date

    seconds = since.seconds + since.microseconds / 1E6 + since.days * 86400

    days = math.floor(seconds / (3600*24))

    if days <= 0 and seconds <= 0:
        # Timezone confusion, is in future
        return "moment ago"

    if days > 7:
        # Full date
```

```
    return date.strftime("%d.%m.%Y %H:%M")
elif days >= 1:
    # Week day format
    return date.strftime("%A %H:%M")
else:
    hours = math.floor(seconds/3600.0)
    minutes = math.floor((seconds % 3600) /60)
    if hours > 0:
        return "%d hours %d minutes ago" % (hours, minutes)
    else:
        if minutes > 0:
            return "%d minutes ago" % minutes
        else:
            return "few seconds ago"
```

Friendly date/time from TAL

From within your TAL templates, you can call `toLocalizedTime()` like:

```
<span tal:replace="python:here.toLocalizedTime(o.ModificationDate)"></span>
```

Sending email

Description

How to programmatically send email in Plone

Introduction

This document tells how to send email from Plone.

Email can be sent:

- manually, by calling *MailHost*;
- using a *Content Rule* (content rules have an email-out action by default) which can be activated by a workflow transition, for example;
- triggering email-based password reset.

Configuring MailHost for a mail queue

`Products.MailHost` supports asynchronous sending in a separate thread via a mail queue.

Note: Using a mail queue is recommended for production sites.

To enable the queue, go to the Management Interface and the MailHost tool. Here, check the “Use mail queue” setting and set the “Queue directory”. The queue directory is given as an absolute path on your server, must have a maildir layout (it needs the directories ‘cur’, ‘new’ and ‘tmp’ in it) and must be writeable by the system user, under which the Zope thread runs.

Manually calling MailHost

After your `mail_text` is prepared, sending it is as simple as:

```
from plone import api

try:
    mail_host = api.portal.get_tool(name='MailHost')
    # The ``immediate`` parameter causes an email to be sent immediately
    # (if any error is raised) rather than sent at the transaction
    # boundary or queued for later delivery.
    return mail_host.send(mail_text, immediate=True)
except SMTPRecipientsRefused:
    # Don't disclose email address on failure
    raise SMTPRecipientsRefused('Recipient address rejected by server')
```

Preparing mail text

`mail_text` can be generated by calling a page template (`.pt`) with keyword arguments. The values are accessed in the template as `option/keyword`. For example, take a sample template:

```
<tal:root define="lt string:&lt;;
                gt string:&gt;;
                dummy python:request.RESPONSE.setHeader('Content-Type', 'text/plain;
↪; charset=%s' % options['charset']);
                member python:options['member'];"
>From: "<span tal:replace="python:here.email_from_name" />" <span tal:replace=
↪"structure lt"/><span tal:replace="python:here.email_from_address" /><span_
↪tal:replace="structure gt"/>
To: <span tal:replace="python:member.getProperty('email')" />
Subject: <span i18n:domain="yourproduct" i18n:translate="yoursubjectline" tal:omit-
↪tag="">Subject Line</span>
Content-Type: text/plain; charset=<span tal:replace="python:options['charset']" />
Dear <span tal:replace="member/getFullname" />:
You can now log in as <span tal:replace="member/getId" /> at <span tal:replace=
↪"python:options['portal_url']" />
Cheers!
The website team
</tal:root>
```

This can be called with a member object and the `portal_url`:

```
mail_template = portal.mail_template_id
mail_text = mail_template(member=member,
                           portal_url=portal.absolute_url(),
                           charset=email_charset,
                           request=REQUEST)
```

For more complete examples (with `i18n` support, etc.) see the password reset modules (particularly `Products.remember.tools.registration`).

Note: If you don't need to have third parties to override your email templates it might be cleaned to use Python string templates, as XML based TAL templates are not designed for plain text templating.

Graceful failing

In the case SMTP server rejects the connection. etc. don't abort the current transaction (which is the default behavior)

- <http://stackoverflow.com/questions/9013009/ploneformgen-and-fail-safe-email-send>

Annotations

Description

How to use annotation design pattern to store arbitrary values on Python objects (Plone site, HTTP request) for storage and caching purposes.

Introduction

Annotations is conflict-free way to stick attributes on arbitrary Python objects.

Plone uses annotations for:

- Storing field data in Archetypes (Annotation storage).
- Caching values on HTTP request object (plone.memoize cache decorators).
- Storing settings information in portal or content object (various add-on products).

See [zope.annotation](#) package.

HTTP request example

Store cached values on HTTP request during the life cycle of one request processing. This allows you to cache computed values if the computation function is called from the different, unrelated, code paths.

```
from zope.annotation.interfaces import IAnnotations

# Non-conflicting key
KEY = "mypackage.something"

annotations = IAnnotations(request)

value = annotations.get(KEY, None)
if value is None:
    # Compute value and store it on request object for further look-ups
    value = annotations[KEY] = something()
```

Content annotations

Overview and basic usage

If you want to extend any Plone content to contain “custom” settings annotations is the recommended way to do it.

- Your add-on can store its settings in Plone site root object using local utilities or annotations.
- You can store custom settings on content objects using annotations.

By default, in content annotations are stored:

- Assigned portlets and their settings.
- Archetypes content type fields using AnnotationStorage (like text field on Document).
- Behavior data from `plone.behavior` package.

Example:

```
# Assume context variable refers to some content item

# Non-conflicting key
KEY = "yourcompany.packagename.magicalcontentnavigationsetting"

annotations = IAnnotations(context)

# Store some setting on the content item
annotations[KEY] = True
```

Advanced content annotation

The above example is enough for storing simple values as annotations. You may provide more complex annotation objects depending on your application logic on various content types. This example shows how to add a simple “Like / Dislike” counter on a content object.

```
class LikeDislike(object):
    def __init__(self):
        self.reset()

    def reset(self):
        self._likes = set()
        self._dislikes = set()

    def likedBy(self, user_id):
        self._dislikes.discard(user_id)
        self._likes.add(user_id)

    def dislikedBy(self, user_id):
        self._likes.discard(user_id)
        self._dislikes.add(user_id)

    def status(self):
        return len(self._likes), len(self._dislikes)
```

At this step it is essential to check that your custom annotation class can be [pickled](#). In the Zope world, this means that you cannot hold in your annotation object any reference to a content too.

Tip: Use the UID of a content object if you need to keep the reference of that content object in an annotation.

The most pythonic recipe to get (and set if not existing) your annotation for a given key is:

```
from zope.annotation import IAttributeAnnotatable, IAnnotations

KEY = 'content.like.dislike' # It's best place is config.py in a real app
```



```
def getLikesDislikeFor(item):
    """Factory for LikeDislike as annotation of a contentish
    @param item: any annotatable object, thus any Plone content
    """
    # Ensure the item is annotatable
    assert IAttributeAnnotatable.providedBy(item) # Won't work otherwise
    annotations = IAnnotations(item)
    return annotations.setdefault(KEY, LikeDislike())
```

This way, you're sure that :

- You won't create annotations on an object that can't support them.
- You will create a new fresh annotation mastered with your LikeDislike for your context object if it does not already exist.
- You can play with your LikeDislike annotation object as with any Python object, all attributes changes will be stored automatically in the annotations of the associated content object.

Wrapping your annotation with an adapter

zope.annotation comes with the factory() function that transforms the annotation class into an adapter (possibly named as the annotation key).

In addition the annotation created this way have location awareness, having __parent__ and __name__ attributes.

Let's go back to the above sample and use the zope.annotation.factory() function.

```
import zope.interface
import zope.component
import zope.annotation

from zope.interface import implements
from zope.annotation import factory

from some.contenttype.interfaces import ISomeContent

KEY = 'content.like.dislike' # It's best place is config.py in a real app

class ILikeDislike(zope.interface.Interface):
    """Model for like/dislike annotation
    """
    def reset():
        """Reinitialize everything
        """

    def likedBy(user_id):
        """User liked the associated content
        """

    def dislikedBy(user_id):
        """User disliked the associated content
        """

class LikeDislike(object):
    implements(ILikeDislike)
    zope.component.adapts(ISomeContent)
```

```
def __init__(self):
    # Does not expect argument as usual adapters
    # You can access annotated object through ``self.__parent__``
    self.reset()

def reset(self):
    self._likes = set()
    self._dislikes = set()

def likedBy(self, user_id):
    self._dislikes.discard(user_id)
    self._likes.add(user_id)

def dislikedBy(self, user_id):
    self._likes.discard(user_id)
    self._dislikes.add(user_id)

def status(self):
    return len(self._likes), len(self._dislikes)

# Register as adapter (you may do this in ZCML too)
zope.component.provideAdapter(factory(LikeDislike, key=KEY))

# Lets play with some content
item = getSomeContentImplementingISomeContent() # Guess what :)

# Let's have its annotation
like_dislike = ILikeDislike(item)

# Play with this annotation
like_dislike.likedBy('joe')
like_dislike.dislikedBy('jane')

assert like_dislike.status() == (1, 1)
assert like_dislike.__parent__ is item
assert like_dislike.__name__ == KEY
```

Tip: Read a full doc / test / demo of the `zope.annotation.factory()` in the `README.txt` file in the root of `zope.annotation` package for more advanced usages.

Cleaning up content annotations

Warning: If you store full Python objects in annotations you need to clean them up during your add-on uninstallation. Otherwise if Python code is not present you can no longer import or export Plone site (annotations are pickled objects in the database and pickles do no longer work if the code is not present).

How to clean up annotations on content objects:

```
def clean_up_content_annotations(portal, names):
    """
```

```

Remove objects from content annotations in Plone site,

This is mostly to remove objects which might make the site un-exportable
when eggs / Python code has been removed.

@param portal: Plone site object

@param names: Names of the annotation entries to remove
"""

output = StringIO()

def recurse(context):
    """ Recurse through all content on Plone site """

    annotations = IAnnotations(context)

    #print >> output, "Recursing to item:" + str(context)
    print annotations

    for name in names:
        if name in annotations:
            print >> output, "Cleaning up annotation %s on item %s" % (name,
↪context.absolute_url())
            del annotations[name]

    # Make sure that we recurse to real folders only,
    # otherwise contentItems() might be acquired from higher level
    if IFolderish.providedBy(context):
        for id, item in context.contentItems():
            recurse(item)

recurse(portal)

return output

```

Make your code persistence free

There is one issue with the above methods: you are creating new persistent classes so your data need your source code. That makes your code hard to uninstall (have to keep the code BBB + cleaning up the DB by walking throw all objects)

Another pattern to store data in annotations: Use already existing persistent base code instead of creating your own.

Please use one of theses:

- BTrees
- PersistentList
- PersistentDict

This pattern is used by `cioppino.twothumbs` and `collective.favoriting` addons.

How to achieve this: <https://gist.github.com/toutpt/7680498>

Other resources

- <https://plone.org/documentation/tutorial/embrace-and-extend-the-zope-3-way/annotations>

Normalizing ids

Description

How to convert arbitrary text input to URL/CSS/file/programming safe ids.

Introduction

Normalizers turns arbitrary string (with unicode letters) to machine friendly ASCII ids. Plone provides different id normalizers.

E.g:

```
åland -> aland
```

Plone has conversion utilities for

- For URIs and URLs (`plone.i18n.normalizer.interfaces.IURLNormalizer`)
- For filenames
- For HTML ids and CSS

Normalization depends on the locale. E.g. in English “æ” will be normalized as “ae” but in Finnish it will be normalized “ä” -> “a”.

See `plone.i18n.normalizers` package.

Examples

Simple example for CSS id:

```
from zope.component import getUtility
from plone.i18n.normalizer.interfaces import IIDNormalizer

normalizer = getUtility(IIDNormalizer)
id = "portlet-static-%s" % normalizer.normalize(header)
```

Hard-coded id localizer which directly uses class instance and does not allow override by utility configuration. You can use normalizers this way also when `getUtility()` is not available (e.g. start up code):

```
from plone.i18n.normalizer import idnormalizer

id = idnormalizer.normalize(u"ÄÄÖrjy")
```

Language specific example for URL:

```

from zope.component import queryUtility
from plone.i18n.normalizer.interfaces import IURLNormalizer

    # Get URL normalizer for language english
util = queryUtility(IURLNormalizer, name="en")

```

To see available language specific localizers, see the source code of `plone.i18n.normalizers` package.

More examples:

- Static text portlets normalizes portlet title for CSS class.

Creating ids programmatically

If you are creating content programmatically using `invokeFactory()` or by calling the class constructor you need to provide the id yourself.

Below is an example how to generate id from a title. *container* is the folderish object that will contain our new object.:

```

import time
import transaction
from zope.container.interfaces import INameChooser

# For the NameChooser to work, it needs our object to already exist.
# We create our object with a temporary but unique id. Seconds since
# epoch will do.
oid = container.invokeFactory(portal_type, id=time.time())

# It's necessary to save the object creation before we can rename it
transaction.savepoint(optimistic=True)
new_obj = container._getOb(oid)

# Now we create and set a new user-friendly id from the object title
title = "My Little Pony"
oid = INameChooser(container).chooseName(title, new_obj)
new_obj.setId(oid)
new_obj.reindexObject()

```

Other

Enforcing normalization for old migrated context.

Monkey-patching

A monkey patch (also spelled monkey-patch, `MonkeyPatch`) is a way to extend or modify the runtime code of dynamic languages (e.g. Smalltalk, JavaScript, Objective-C, Ruby, Perl, Python, Groovy, etc.) without altering the original source code.

Plone community promotes conflict free way to do monkey patching using `collective.monkeypatcher` package.

Patching constants

Some modules (typically `config.py` files) include constant definitions used throughout the package. Given that `collective.monkeypatcher` is intended to patch methods you'll not be able to patch a constant straightforward.

Instead you'll have to make use of the handler option:

```
<monkey:patch
  description="Add new terabyte constant"
  class="Products.CMFPlone.CatalogTool.CatalogTool"
  original="SIZE_CONST"
  replacement=".patches.patched_size_const"
  handler=".patches.apply_patched_const"
/>
```

And your `patches.py` module should include this:

```
NEW_SIZE_CONST = {'kB': 1024, 'MB': 1024*1024, 'GB': 1024*1024*1024, 'TB':
↪1024*1024*1024*1024}

patched_size_const = lambda : NEW_SIZE_CONST # Now we have a callable method!

def apply_patched_const(scope, original, replacement):
    setattr(scope, original, replacement())
    return
```

This way the **original** `SIZE_CONST` constant would be replaced by the result of the lambda function, which is our new constant.

Patching @property methods

If you are to patch a @property decorated method you can use the handler configuration option:

```
<monkey:patch
  description="Performance boost in foldercontents"
  class="plone.app.content.browser.foldercontents.FolderContentsTable"
  original="items"
  replacement=".patches.patched_items"
  handler=".patches.apply_patched_property"
/>
```

And your `patches.py` module should include this:

```
def items(self):
    ... # The body of your patched method

def apply_patched_property(scope, original, replacement):
    # This is actually the same as apply_patched_const above
    setattr(scope, original, replacement())
    return

patched_items = lambda : property(items) # We get a @property decorated method!
```

This way the **original** `items` method would be replaced by the result of the lambda function, which is a @property decorated method written in a different way.

Command-line interaction and scripting

Description

How to run command-line Python scripts, timed jobs (cron) and batch jobs against Plone sites and Zope application server.

- *Introduction*
- *Starting interactive interpreter*
- *Running scripts*
- *Cron and timed jobs*
- *Scripting context*
- *Committing transactions*
- *zopepy*
- *Setting up ZEO for command line-processing*
- *Posing as user*
- *Spoofing HTTP request*
- *Creating Plone site in buildout*
- *screen*
 - *Start new screen*
 - *Attach to an existing screen*

Introduction

Warning: Plone code is somewhat ugly and expects you to have real HTTP request lifecycle to do many things. For command line scripts, you need to mock up this and mocking up often fails. Instead of trying to create a pure command-line script, just create a browser view and call that browser view usings wget or lynx or similar command line HTTP tool.

Zope provides facilities to run command-line scripts. or maintenance work, like migration script.

- The output to terminal is instance (Plone buffers HTML output)
- You can stop processing using CTRL+C
- You can integrate scripts with standard UNIX tools, like cron

Note: If the site runs in a single process Zope mode (no ZEO), the actual site instance must be stopped to run a command line script as the one process locks the database (Data.fs).

Command line scripts are also useful for long-running transaction processing

- A web site runs in multi-client ZEO mode. One client is always offline, reserved for running command-line scripts.
- Asynchronous long-running transactions are run from this ZEO client, without disturbing the normal site functionality

See also

- [lovely.remotetask package](#) for more fine-grained control and Zope-based cron jobs

Starting interactive interpreter

The `bin/instance debug` command starts an interactive interpreter with the Zope application server and database loaded. You can provide the id of your Plone site with the `-O` flag to have it available under the name *obj* and to load **:doc:'persistent utilities </develop/addons/components/utilities>'**_. The following example assumes the site is named “Plone”. For more infos about command-line options use `bin/instance help debug`.

Example:

```
bin/instance -OPlone debug
```

Note: The instance must be stopped in order to run this.

Running scripts

Use `bin/instance run` command to run scripts which can interact with the opened database.

Example:

```
bin/instance run src/namespace.mypackage/namespace/mypackage/bin/script.py
```

The script will have global `app` variable assigned to the Zope application server root. You can use this as a starting point and traverse into your Plone site(s).

Script could look like:

```
"""
    Instance script for testing a researcher creation

    Execution::

        bin/instance run src/x.y/x/y/testscript.py
"""

from ora.objects.content.oraresearcher import createResearcherById

def main(app):
    folder = app.unrestrictedTraverse("x/y/z/cancer")

    # Create a researcher
    createResearcherById(folder, "http://localhost/people/9947603276956765")

    # This script does not commit

# If this script lives in your source tree, then we need to use this trick so that
# five.grok, which scans all modules, does not try to execute the script while
# modules are being loaded on the start-up
if "app" in locals():
    main(app)
```

You probably need to spoof your *security credentials*.

Note: Instance must be stopped in order to run this.

Cron and timed jobs

Cron is UNIX clock daemon for timed tasks.

If you have a ZEO cluster you can have one ZEO client reserved for command line processing. Cron job will run scripts through this ZEO client.

Alternatively, you can use

- cron to call localhost URL using curl or wget UNIX commands
- Use Zope clock daemon

Note: For long running batch processes it is must that you run your site in ZEO mode. Otherwise the batch job will block the site access for the duration of the batch job transaction. If the batch job takes long to process the site might be unavailable for the visitors for a long period.

Scripting context

The command line interpreter and scripts gets following global context variables

- *app* global variable which holds the root of Zope application server.
- *sys.argv* contains command-line parameters after python script name
 - *argv[0]* = script name
 - *argv[1]* = first command line argument

To access your site object, you can traverse down from *app*:

```
app.yoursiteid # This is your Plone site object

# Perform some stuff here...
for brain in app.yoursiteid.portal_catalog(portal_type="Document"): print brain["Title"]
```

Committing transactions

You need to manually commit transactions if you change ZODB data from the command line.

Example how to commit:

```
# Commit transaction
import transaction; transaction.commit()
# Perform ZEO client synchronization (if running in clustered mode)
app._p_jar.sync()
```

More info

- <http://www.enfoldsystems.com/software/server/docs/4.0/enfolddebuggingtools.html>

zopepy

zopepy buildout recipe creating bin/zopepy command which you can use to run Python scripts in Zope environment set-up (PYTHONPATH, database connection, etc.)

- <https://pypi.python.org/pypi/zc.recipe.egg>

buildout.cfg example:

```
[zopepy]
# For more information on this step and configuration options see:
#
recipe = zc.recipe.egg
eggs = ${client1:eggs}
interpreter = zopepy
extra-paths = ${zope2:location}/lib/python
scripts = zopepy
```

Then running:

```
bin/zopepy path/to/myscript.py
```

...or if you want to run a script outside buildout folder:

```
cd /tmp
/srv/plone/site/bin/zopepy pack2.py
```

Setting up ZEO for command line-processing

Plone site HTTP requests are processed by one process per requests. One process cannot handle more than one request once. If you need to have long-running transactions you need to at least two front end processes, ZEO clients, so that long-running transactions won't block your site.

- [Converting instance to ZEO based configuration](#)

Your code might want to call `transaction.commit()` now and then to commit the current transaction.

Posing as user

Zope functionality often assumes you have logged in as certain user or you are anonymous user. Command-line scripts do not have user information set by default.

How to set the effective Zope user to a regular user using plone.api context managers:

```
from plone import api
from zope.component.hooks import setSite

# Sets the current site as the active site
setSite(app['Plone'])

# Enable the context manager to switch the user
with api.env.adopt_user(username="admin"):
```

```
# You're now posing as admin!
portal.restrictedTraverse("manage_propertiesForm")
```

Spoofing HTTP request

When running from command-line, HTTP request object is not available. Some Zope code might expect this and you need to spoof the request.

Below is an example command line script which set-ups faux HTTP request and portal_skins skin layers:

```
"""
    Command-line script to be run from a ZEO client:

    bin/command-line-client src/yourcode/mirror.py
"""

import os
from os import environ
from StringIO import StringIO
import logging

from AccessControl.SecurityManagement import newSecurityManager
from AccessControl.SecurityManager import setSecurityPolicy
from Testing.makerequest import makerequest
from Products.CMFCore.tests.base.security import PermissiveSecurityPolicy, \
    OmnipotentUser

# Force application logging level to DEBUG and log output to stdout for all loggers
import sys, logging

root_logger = logging.getLogger()
root_logger.setLevel(logging.DEBUG)

handler = logging.StreamHandler(sys.stdout)
formatter = logging.Formatter("%(asctime)s - %(name)s - %(levelname)s - %(message)s")
handler.setFormatter(formatter)
root_logger.addHandler(handler)

def spoofRequest(app):
    """
        Make REQUEST variable to be available on the Zope application server.

        This allows acquisition to work properly
    """
    _policy=PermissiveSecurityPolicy()
    _oldpolicy=setSecurityPolicy(_policy)
    newSecurityManager(None, OmnipotentUser().__of__(app.acl_users))
    return makerequest(app)

# Enable Faux HTTP request object
app = spoofRequest(app)

# Get Plone site object from Zope application server root
```

```
site = app.unrestrictedTraverse("yoursiteid")
site.setupCurrentSkin(app.REQUEST)

# Call External Method defined in the skins layers
# Note that native python __getattr__ traversing does not work... you must access_
↳things using unrestrictedTraverse()
# You could also use @@viewname for browserviews
script = site.unrestrictedTraverse("someScriptName")
script()
```

More info

- <http://wiki.zope.org/zope2/HowToFakeREQUESTInDebugger>

Creating Plone site in buildout

You can pre-generate the site from the buildout run.

- <https://pypi.python.org/pypi/collective.recipe.plonesite#example>

screen

screen is an UNIX command to start a virtual terminal. Screen lets processes run even if your physical terminal becomes disconnected. This effectively allows you to run long-running command line jobs over a crappy Internet connection.

Start new screen

Type command:

```
screen
```

If you have sudo'ed to another user you first need to run:

```
script /dev/null
```

- <http://dbadump.blogspot.com/2009/04/start-screen-after-sudo-su-to-another.html>

Attach to an existing screen

Type command:

```
screen -x
```

Description

Interacting with Plone from the Zope debug prompt.

Introduction

The Zope debug prompt is a way of starting a Zope client from the command line, and allows you to directly interact with Plone.

Some things that you can do through the debug prompt:

- Examine objects to see their properties/methods
- Update objects in bulk, as opposed to one at a time through the user interface.
- Produce reports or export data from objects

This is very similar to the Python debug prompt (just entering `python` at the command line) and the same whitespace restrictions apply.

Cautions

With great power comes great responsibility.

Interacting with Plone from the Zope debug prompt is a very powerful tool, and lets you quickly make changes that might take hours or days to implement manually.

It is also a great way to severely damage your site, in a way that might take hours or days to fix.

Precautions for developing code that makes updates via the debug prompt:

- Log the changes that you're going to make using the `print` statement
- Only commit the transaction (`transaction.commit()`) after the code has run successfully.
- Attempt to develop the code in an idempotent (able to be run multiple times with no ill effects) manner.

Starting the debug prompt interactively

This assumes that you are running Plone in a ZEO configuration on a *NIX server, and your Zope clients run as the `plone_daemon` user.

After logging into your server, start the debug prompt with:

```
sudo -u plone_daemon /path/to/zope/bin/client1 debug
```

The output will look something like:

```
Starting debugger (the name "app" is bound to the top-level Zope object)
2015-06-26 12:33:51 WARNING SecurityInfo Conflicting security declarations for
↳ "manage_pasteObjects"
2015-06-26 12:33:51 WARNING SecurityInfo Class "CopyContainer" had conflicting
↳ security declarations
2015-06-26 12:33:52 WARNING Init Class Products.Five.metaclass.RedirectsView has a
↳ security declaration for nonexistent method 'errors'

>>>
```

There may be some additional warnings, based on the products installed.

From here, the `app` variable is equivalent to the root of your Zope instance.

A simple example:

```
from AccessControl.SecurityManagement import newSecurityManager
from Testing.makerequest import makerequest
from zope.component.hooks import setSite
from plone import api
import transaction

app = makerequest(app) # Enables functionality that expects a REQUEST

app._p_jar.sync() # Syncs the debug prompt with any transactions

site=app['Plone'] # Use your site id

setSite(site) # Sets the current site as the active site

# Simulate logging in as the Zope 'admin' user
with api.env.adopt_user(username="admin"):
    # Grab the portal_catalog tool
    portal_catalog = api.portal.get_tool(name='portal_catalog')

    # Query the catalog for all results. Returns 'brains'
    results = portal_catalog.searchResults()

    # Print the number of objects in the catalog
    print u"My site has %d objects." % len(results)
```

Code Snippets

Sample code snippets for use in the debug prompt.

Clock and asynchronous tasks

Description

How to run background tasks or cron jobs with Zope

Cron jobs

You can use simple UNIX cron + wget combo to make timed jobs in Plone.

If you need to authenticate, e.g. as an admin, Zope users (not Plone users) can be authenticated using HTTP Basic Auth.

- Create user in Zope root (not Plone site root) in acl_users folder
- Call it via HTTP Basic Auth

`http://username:password@localhost:8080/yoursiteid/@@clock_view_name`

- The `--auth-no-challenge` option to the wget command will authenticate even if the server doesn't ask you to authenticate. It might come in handy, as Plone does not ask for HTTP authentication, and will just serve Unauthorized if permissions aren't sufficient.

Clock server

You can make Zope to make regular calls to your views.

Add in buildout.cfg:

```
zope-conf-additional =
    <clock-server>
        method /xxx/feed-mega-update
        period 3600
        user zopeuser
        password 123123
        host xxx.com
    </clock-server>

    <clock-server>
        method /yyy/feed-mega-update
        period 3600
        user zopeuser
        password 123123
        host yyy.com
    </clock-server>
```

Create a corresponding user in Management Interface.

In detail:

- method - Path from root to an executable Zope method (Python script, external method, etc.) The method must receive no arguments.
- period - Seconds between each call to the method. Typically, at least 30 specified.
- user - a Zope Username
- password - The password of this user Zope
- host - The name of the host that is in the header of a request as host: is specified.

To check whether the server clock is running, restart the instance or the ZEO client in the foreground and see if a message similar to the following is displayed:

```
2009-03-03 19:57:38 INFO ZServer Clock server for "/ mysite / do_stuff" started_
↪(user: admin, period: 60)
```

If you are using a public source control repository for your buildout.cfg you might want to put zope-conf-additional= to secret.cfg which lies only on the production server and is never committed to the version control:

```
# Change the number here to change the version of Plone being used
extends =
    http://dist.plone.org/release/4.1rc3/versions.cfg
    http://good-py.appspot.com/release/dexterity/1.0?plone=4.1rc3
    http://plonegomobile.googlecode.com/svn/gomobile/buildout/gomobile.plone-4.
↪trunk.commit-access.cfg
    secret.cfg
```

Creating a separate ZEO instance for long running tasks

Below is an example how to extend a single process Plone instance buildout to contain two ZEO front end processes, client1 and client2 and dedicate client2 for long running tasks. In this example, Products.feedfeeder RSS

zopeuser is set to run on client2.

- Client1 keeps acting like standalone instance, in the same port as instance used to be
- Clocked tasks are run on client2 - it does not serve other HTTP requests. Clocked tasks are done using Zope clock server.

The purpose of this is that client2 does heavy writes to the database, potentially blocking the normal site operation of the site if we don't have a separate client for it.

We create additional `production.cfg` file which extends the default `buildout.cfg` file. You still can use `buildout.cfg` as is for the development, but on the production server your buildout command must be run for the ZEO server enabled file.

Actual clock server jobs, with usernames and passwords, are stored in a separate `secret.cfg` file which is only available on the production server and is not stored in the version control system. The user credentials for a specially created Zope user, not Plone user. This user can be created through `acl_users` in the Management Interface.

We also include `plonectl` command for easy management of ZEO server, client1 and client2.

`production.cfg` (note - you need to run this with `bin/buildout -c production.cfg`):

```
[buildout]

extends =
    buildout.cfg
    secret.cfg

# Add new stuff to be build out when run bin/buildout -c production.cfg

parts +=
    client1
    client2
    zeoserver
    plonectl
    crontab_zeopack

# Run our database and stuff
[zeoserver]
recipe = plone.recipe.zeoserver
zeo-address = 9998

# In ZEO server mode, client1 is clone of standalone
# [instance] running in ZEO mode, different port
[client1]
<= instance
recipe = plone.recipe.zope2instance
zeo-client = on
shared-blob = on
http-address = 9999

# Client2 is like client1, just different port.
# This client is reserved for running clocked tasks (feedfeeder update)
[client2]
<= client1
http-address = 9996

# Tune down cache-size as we don't operate normally,
# so we have smaller memory consumption (default: 10000)
zodb-cache-size = 3000
```



```
[plonectl]
recipe = plone.recipe.unifiedinstaller
clients =
    client1
    client2
user = admin:admin

# pack your ZODB each Sunday morning and hence make it smaller and faster
[crontab_zeopack]
recipe = z3c.recipe.usercrontab
times = 0 1 * * 6
command = ${buildout:directory}/bin/zeopack
```

`secret.cfg` contains actual clocked jobs. This file contains passwords so it is not recommended to put it under the version control:

```
[client2]
zope-conf-additional =
    <clock-server>
        method /plonecommunity/feed-mega-update
        period 3600
        user zopeuser
        password secret
        host plonecommunity.mobi
    </clock-server>

    <clock-server>
        method /plonecommunity/@@feed-mega-cleanup?days=14
        period 85000
        user zopeuser
        password secret
        host plonecommunity.mobi
    </clock-server>

    <clock-server>
        method /mobipublic/feed-mega-update
        period 3600
        user zopeuser
        password secret
        host mobipublic.com
    </clock-server>

    <clock-server>
        method /mobipublic/@@feed-mega-cleanup?days=14
        period 84000
        user zopeuser
        password secret
        host mobipublic.com
    </clock-server>

    <clock-server>
        method /mobipublic/find-it/events/@@event-cleanup?days=1
        period 84000
        user zopeuser
        password secret
        host mobipublic.com
    </clock-server>
```

Asynchronous

Asynchronous tasks are long-running tasks which are run on their own thread.

lovely.remotetask

`lovely.remotetask` is worked based long-running task manager for Zope 3.

- [lovely.remotetask package package page](#)
- <http://tarekziade.wordpress.com/2007/09/28/a-co-server-for-zope/>
- <http://swik.net/Zope/Planet+Zope/Trying+lovely.remotetask+for+cron+jobs/c1kfs>
- <http://archives.free.net.ph/message/20081015.201535.2d147fec.fr.html>

Flowplayer

Description

Using Flowplayer video player in your Plone add-ons.

Introduction

Flowplayer is a GPL'ed Flash-based video player.

Plone integration exists as an add-on product:

- <https://plone.org/products/collective-flowplayer>

Creating a custom Flowplayer

Here is a walkthrough how to create a custom content type with a video field which plays the uploaded video using Flowplayer in a page template with parameters you define.

Dexterity model definition:

```
from plone.namedfile.field import NamedFile, NamedImage

class IPortletSource(form.Schema):
    """ Portlet source content
    """

    videoFile = NamedFile(
        title=u"Video file",
        description=u"Upload video file from local computer to show mini-
↪video-player in the portlet",
        required=False
    )
```

Helper view Python code:

```

class MiniVideo(grok.View):
    """ Render a mini Flowplayer (inside portlet)
    """

    grok.context(IPortletSource)

    def hasVideo(self):
        """ Check if plone.namedfile field exist and has a video file uploaded on the
        ↪context item.
        """

        return self.context.videoFile != None

```

Helper view template:

```

<div class="video" tal:condition="view/hasVideo">

    <!-- The href references the FLV file. It is not safe to use XHTML
    style self-closing tags here. -->
    <tal:video define="video nocall:context/videoFile"
        tal:condition="nocall:video">
        <a class="flow-player" tal:attributes="href string:${context/absolute_url}
        ↪/@@download/videoFile/${video/filename}">
            </a>
        </tal:video>

    <!-- Helper for JavaScript which is used to determine location of
    Flowplayer resource files -->
    <span class="flowplayer-site-url" style="display:none" tal:content="context/
    ↪portal_url" />

</div>

```

Using the view:

```

<div class="portletSimulator">

    <div tal:attributes="class string:ls-portlet ${context/extraCSS}">

        <h3 tal:condition="view/has_title"
            tal:attributes="class string:portletHeader ls-portlet-header">
            <a class="header"
                tal:omit-tag=""
                tal:content="context/title" />
            </h3>

        <div tal:define="videoView nocall:context/@@minivideo"
            tal:replace="structure videoView" />

        </div>

    </div>

</div>

```

JavaScript, registered in portal_javascripts, doing the magic:

```

/**
 * Bootstrap flow player.
 *

```

```
* Call this when DOM is ready ( jq(document).ready() ).
*/
function setupPortletVideo() {

    // Site base URL must be available in some hidden variable
    // so that we can build references to our media resources
    var urlBase = jq(".flowplayer-site-url").text();

    console.log("Video set-up:" + urlBase);

    // Iterate through all links which are tagged as video on the page
    // Use a special marker class for videos which we want to configure ourselves
    jq('a.flow-player').each(function() {

        console.log("Found flowplayer");

        var self = jq(this);

        // Config help
        // http://flowplayer.org/documentation/configuration/index.html
        // http://flowplayer.org/documentation/configuration/clips.html#properties
        // Styling properties http://flowplayer.org/documentation/configuration/
        ↪ plugins.html
        var config = {
            "clip": {
                "scaling": "original",
                "autoBuffering": true,
                "autoPlay": false,
            },

            "plugins": {
                // Note that + must be escaped as %2B
                "audio": {
                    "url": urlBase + "%2B%2Bresource%2B%2Bcollective.flowplayer/"
                    ↪ flowplayer.audio.swf },
                // Disable control plug-in
                // On mouse over Play button still appears
                "controls" : {
                    "url": urlBase + "%2B%2Bresource%2B%2Bcollective.flowplayer/"
                    ↪ flowplayer.controls.swf,
                    playlist:false,
                    fullscreen:false,
                    mute:false,
                    time:false,
                },

                // http://flowplayer.org/documentation/configuration/player.html
                // debug : true,

                log: {
                    // Enable debug output (lots of it)
                    // level : 'debug'
                },

            }

            config.clip.url = self.attr('href');
```

```
// Create Flowplayer by calling its own JS API
var player = flowplayer(this,
    {"src": urlBase + "++resource++collective.flowplayer/flowplayer.swf"
    }, config);
});
}

jq(document).ready(setupPortletVideo);
```

Needed CSS:

```
/* Videos */

a.flow-player {
    display: block;
    width: 235px;
    height: 180px;
}
```

Note: if your player is not displayed on the page load, but is displayed after you click somewhere to the player container area, be sure there is no HTML code nor text inside the player container HTML tag. Such code/text is considered as player splash screen and player is waiting for click to the splash.

Non-buffered MP4 playback fix

MPEG4 files must be specially prepared (quick play fix), so that the playback starts instantly and the player does not try to buffer the whole file first

- <https://twitter.com/moo9000/status/253947688276594688>

Navigation trees

Description

How navigation trees are generate in Plone and how to generate custom navigation trees.

Introduction

Plone exposes methods to build navigation trees.

- `Products.CMFPlone.browser.navtree`
- `plone.app.layout.navigation.navtree.buildFolderTree`

These are internally used by navigation portlet and sitemap.

Creating a custom navigation tree

See `Products.PloneHelpCenter` for full code.

The following example builds Table of Contents for *Reference Manual* content type:

```
class Strategy(NavtreeStrategyBase):

    rootPath = '/'.join(root.getPhysicalPath())
    showAllParents = False

strategy = Strategy()
query= {'path'      : '/'.join(root.getPhysicalPath()),
        'object_provides' : 'Products.PloneHelpCenter.interfaces.IHelpCenterMultiPage
↪',
        'sort_on'      : 'getObjPositionInParent'}

toc = buildFolderTree(self, current, query, strategy)['children']
```

Excluding items in the navigation tree

Your navigation tree strategy must define method `nodeFilter()` which can check for `portal_catalog` metadata column `exclude_from_nav`.

Example (from `Products.CMFPlone.broser.navtree`):

```
class SitemapNavtreeStrategy(NavtreeStrategyBase):

    def nodeFilter(self, node):
        item = node['item']
        if getattr(item, 'exclude_from_nav', False):
            return False
        else:
            return True
```

Querying items in natural sort order

Sometimes you want to display content items as they appear in Plone navigation. Below is an example which builds a flat vobulary for a form checkbox list based on a custom `portal_catalog` query and root folder.

`query_items_in_natural_sort_order`:

```
from plone.app.layout.navigation.navtree import buildFolderTree
from plone.app.layout.navigation.navtree import NavtreeStrategyBase
# https://github.com/plone/Plone/blob/master/Products/CMFPlone/browser/
↪navtree.py
from Products.CMFPlone.browser.navtree import SitemapNavtreeStrategy,
↪DefaultNavtreeStrategy

def query_items_in_natural_sort_order(root, query):
    """
    Create a flattened out list of portal_catalog queried items in their natural_
    ↪depth first navigation order.

    @param root: Content item which acts as a navigation root

    @param query: Dictionary of portal_catalog query parameters

    @return: List of catalog brains
```

```

"""

# Navigation tree base portal_catalog query parameters
applied_query= {
    'path' : '/'.join(root.getPhysicalPath()),
    'sort_on' : 'getObjPositionInParent'
}

# Apply caller's filters
applied_query.update(query)

# Set the navigation tree build strategy
# - use navigation portlet strategy as base
strategy = DefaultNavtreeStrategy(root)
strategy.rootPath = '/'.join(root.getPhysicalPath())
strategy.showAllParents = False
strategy.bottomLevel = 999
# This will yield out tree of nested dicts of
# item brains with retrofitted navigational data
tree = buildFolderTree(root, root, query, strategy=strategy)

items = []

def flatten(children):
    """ Recursively flatten the tree """
    for c in children:
        # Copy catalog brain object into the result
        items.append(c["item"])
        children = c.get("children", None)
        if children:
            flatten(children)

flatten(tree["children"])

return items

```

How to use:

```

def make_terms(items):
    """ Create zope.schema terms for vocab from tuples """
    terms = [ SimpleTerm(value=pair[0], token=pair[0], title=pair[1]) for pair in_
↪items ]
    return terms

def course_source(context):
    """
    Populate vocabulary with values from portal_catalog.

    @param context: z3c.form.Form context object (in our case site root)

    @return: SimpleVocabulary containg all areas as terms.
    """

    # Get site root from any content item using portal_url tool thru acquisition
    root = context.portal_url.getPortalObject()

    context = root.unrestrictedTraverse("courses")

```

```
# We need to include "Folder" in the query even if it's not any of the results -
# this is because the query criteria must match the root content item too
brains = query_items_in_natural_sort_order(context, query = { "portal_type" : [
↪ "xxx2011.app.courseinfo", "xxx2011.app.subjectgroup", "xxx2011.app.coursecategory",
↪ "Folder"] })

def filter(brain):
    # Remove some unwanted items from the list
    # XXX: Not needed anymore after new content types - remove
    x = brain["Title"]

    if "Carousel" in x:
        return False

    return True

# Create a list of tuples (UID, Title) of results
result = [ (brain["UID"], brain["Title"]) for brain in brains if filter(brain) ==
↪ True ]

# Convert tuples to SimpleTerm objects
terms = make_terms(result)

return SimpleVocabulary(terms)

directlyProvides(course_source, IContextSourceBinder)
```

Slidehows and carousels

Description

How to use annotation design pattern to store arbitrary values on Python objects (Plone site, HTTP request) for storage and caching purposes.

Introduction

Header slideshows

- [Products.Carousel](#)

AJAX'y image pop-ups

- <https://plone.org/products/pipbox>

Migrate Products.Slideshow to Products.Carousel

Here is a sample migration code to transform your site from one add-on to another.

We create a migration view which you can call by typing in view name manually to web browser.

This code will

- Scan site for folders which have Slideshow add-on enabled. In this example we check against a predefined list (scanned earlier), but the code contains example how to detect slideshow folders

- Create Carousel for those folders
- Create corresponds Carousel Banners for all Slideshow Image content items
- Set some Carousel settings
- Make sure that we invalidate cache for content items going through migration
- Set a new default view for folders which were using slideshow

Also

- After inspecting the process was ok you can delete migrated images

carousel.py:

```
"""
    Migrate slideshow to carousel.

    Usage:

    http://yoursite/@@migrate_carousel - the process can be repeated with adjusted_
    ↪ settings. It's non-destructive.

    http://yoursite/@@delete_migrated_slideshow_images
"""

import logging
from StringIO import StringIO
from Products.Five.browser import BrowserView

from zope.component import getUtility, getMultiAdapter
from zope.app.component.hooks import setHooks, setSite, getSite

from zope.interface import alsoProvides
from Products.Five import BrowserView
from Products.CMFCore.utils import getToolByName
from Products.Carousel.interfaces import ICarouselFolder
from Products.Carousel.utils import addPermissionsForRole
from Products.Carousel.config import CAROUSEL_ID
from Products.Carousel.interfaces import ICarousel, ICarouselSettings

from Products.slideshowfolder.interfaces import ISlideShowSettings, ISlideShowView,
    ↪ IFolderSlideShowView, ISlideShowFolder, ISlideShowImage

logger = logging.getLogger("Slideshow Migrator")

FOLDER_PATHS_TO_MIGRATE="""
(' ', 'site', 'folder1')
(' ', 'site', 'folder2')
(' ', 'site', 'folder2', 'subfolder')
"""

class MigrateSlideshowToCarousel(BrowserView):
    """
    Migrate collective.slideshow to Products.Carousel
    """
```

```
def startCapture(self, newLogLevel = None):
    """ Start capturing log output to a string buffer.

    http://docs.python.org/release/2.6/library/logging.html

    @param newLogLevel: Optionally change the global logging level, e.g. logging.
    ↪DEBUG """
    self.buffer = StringIO()

    print >> self.buffer, "Log output"

    rootLogger = logging.getLogger()

    if newLogLevel:
        self.oldLogLevel = rootLogger.getEffectiveLevel()
        rootLogger.setLevel(newLogLevel)
    else:
        self.oldLogLevel = None

    self.logHandler = logging.StreamHandler(self.buffer)
    formatter = logging.Formatter("%(asctime)s - %(name)s - %(levelname)s -
    ↪%(message)s")
    self.logHandler.setFormatter(formatter)
    rootLogger.addHandler(self.logHandler)

def stopCapture(self):
    """ Stop capturing log output.

    @return: Collected log output as string
    """

    # Remove our handler
    rootLogger = logging.getLogger()

    # Restore logging level (if any)
    if self.oldLogLevel:
        rootLogger.setLevel(self.oldLogLevel)

    rootLogger.removeHandler(self.logHandler)

    self.logHandler.flush()
    self.buffer.flush()

    return self.buffer.getvalue()

def getImages(self, folder):
    """ Get all ATImages in a folder """
    for obj in folder.objectValues():
        if obj.portal_type == "Image":
            yield obj

def getOrCreateCarousel(self, folder):
    """ Copied from Products.Carousel.manager.py """

    if hasattr(folder.aq_base, CAROUSEL_ID):
```

```

        logger.info("Using existing carousel in " + str(folder))
        carousel = getattr(folder, CAROUSEL_ID)
    else:
        logger.info("Creating carousel in " + str(folder))
        pt = getToolByName(folder, 'portal_types')
        newid = pt.constructContent('Folder', folder, 'carousel', title='Carousel_
↪Banners', excludeFromNav=True)
        carousel = getattr(folder, newid)

        # mark the new folder as a Carousel folder
        alsoProvides(carousel, ICarouselFolder)

        # make sure Carousel banners are addable within the new folder
        addPermissionsForRole(carousel, 'Manager', ('Carousel: Add Carousel Banner
↪',))

        # make sure *only* Carousel banners are addable
        carousel.setConstrainTypesMode(1)
        carousel.setLocallyAllowedTypes(['Carousel Banner'])
        carousel.setImmediatelyAddableTypes(['Carousel Banner'])

    return carousel

def imageToCarouselBanner(self, image, carousel):
    """
    Convert ATImage to Carousel Banner content item.
    """

    logger.info("Migrating slideshow image:" + str(image.getId()))

    id = image.getId()

    if not id in carousel.objectIds():
        carousel.invokeFactory("Carousel Banner", id, title=image.Title())
    else:
        logger.info("Carousel image already existed " + str(image))

    banner = carousel[id]

    # Copy over image field from ATImage content type
    banner.setImage(image.getImage())

    # Set a hidden flag which allows us later to delete images
    image._migrated_to_carousel = True

    from Products.CMFCore.WorkflowCore import WorkflowException

    workflowTool = getToolByName(banner, "portal_workflow")
    try:
        workflowTool.doActionFor(banner, "publish")
        logger.info("Published " + banner.getId())
    except WorkflowException:
        # a workflow exception is risen if the state transition is not available
        # (the sampleProperty content is in a workflow state which
        # does not have a "submit" transition)
        logger.info("Could not publish:" + str(banner.getId()) + " already_
↪published?")

```

```
pass

def setupCarousel(self, carousel_folder):
    """
    Set-up custom carousel settings for all carousels.
    """

    logger.info("Setting carousel settings for:" + carousel_folder.absolute_url())

    settings = ICarouselSettings(carousel_folder)

    settings.width = 640
    settings.height = 450
    settings.pager_template = u'@@pager-none'
    settings.default_page_only = False
    settings.element_id = "karuselli"
    settings.transition_delay = 5.0
    settings.banner_elements = [ u"image" ]

def migrateFolder(self, folder):
    """ Migrate one folder from Slideshow to Products.Carousel
    """
    logger.info("Migrating folder:" + str(folder))

    carousel = self.getOrCreateCarousel(folder)

    self.setupCarousel(carousel)

    images = self.getImages(folder)
    for image in images:
        self.imageToCarouselBanner(image, carousel)

    # This will toggle cache refresh for the object
    # if Products.CacheSetup is used -> should invalidate template cache.
    # Not necessary if Products.CacheSetup is not installed.
    folder.setTitle(folder.Title())
    folder.reindexObject()

    # Toggle folder away from slideshow view
    # empty_view is our custom view which does not list folder contents
    folder.setLayout("empty_view")

    # Set a marker flag in the case we need to play around with these
    # folders programmatically in the future
    folder._migrated_to_carousel = True

def migrate(self):
    """
    Run the migration process for one Plone site.
    """

    brains = self.context.portal_catalog(portal_type="Folder")

    # Use predefined report of slideshow folder on old site
```

```

# Alternative: detect slideshow folders as shown below
carousel_folders = FOLDER_PATHS_TO_MIGRATE.split("\n")

for b in brains:

    obj = b.getObject()

    path = str(obj.getPhysicalPath())

    # Alternative: if you don't have fixed list check here if getattr(obj,
→ "default_view", "") == "slideshow_view"
    if path in carousel_folders:
        self.migrateFolder(obj)

def __call__(self):
    """ Process the form.

    Process the form, log the output and show the output to the user.
    """

    self.logs = None

    try:
        self.startCapture(logging.DEBUG)

        logger.info("Starting full site migration")

        # Do the long running,
        # lots of logging stuff
        self.migrate()

        logger.info("Successfully done")

    except Exception, e:
        # Show friendly error message
        logger.exception(e)

        # Put log output for the page template access
        self.logs = self.stopCapture()

    return self.logs

class DeleteMigratedImages(BrowserView):
    """
    Delete all slideshow image files which have been migrated to carousel banners.

    By doing migration in two phases allows us to adjust the process in the case it
→ goes wrong.
    """

    def __call__(self):
        """

        """

```

```
self.buffer = StringIO()

print >> self.buffer, "Log output"

brains = self.context.portal_catalog(portal_type="Image")
for b in brains:
    obj = b.getObject()
    if getattr(obj, "_migrated_to_carousel", False) == True:
        print >> self.buffer, "Deleting migrated Image " + obj.getId()
        id = obj.getId()
        parent = obj.aq_parent
        parent.manage_delObjects([id])

print >> self.buffer, "All migrated images deleted"

return self.buffer.getvalue()
```

ZCML bits:

```
<browser:page
  for="*"
  name="migrate_carousel"
  permission="cmf.ManagePortal"
  class=".carousel.MigrateSlideshowToCarousel"
/>

<browser:page
  for="*"
  name="delete_migrated_slideshow_images"
  permission="cmf.ManagePortal"
  class=".carousel.DeleteMigratedImages"
/>
```

Setting every carousel widths on the site

Another example to manipulate Products.Carousel. This script will update all carousel settings on the site to have new image width.

```
class SetCarouselWidths(BrowserView):
    """
    Set width to all carousels on the site.
    """

    def __call__(self):
        """

        self.buffer = StringIO()

        print >> self.buffer, "Log output"

        brains = self.context.portal_catalog(portal_type="Folder")
        for b in brains:
            obj = b.getObject()
            if "carousel" in obj.objectIds():
```

```
        carousel = obj["carousel"]
        # Carousel installed on this folder
        settings = ICarouselSettings(carousel)
        print >> self.buffer, "Setting width for " + carousel.absolute_url()
        settings.width = 680

    print >> self.buffer, "All carousels updated"

    return self.buffer.getvalue()
```

ZCML

```
<browser:page
    for="*"
    name="set_carousel_widths"
    permission="cmf.ManagePortal"
    class=".carousel.SetCarouselWidths"
/>
```

AJAX full-size image loading for album views

Plone album views can be converted to pop-up image viewing with PipBox.

Put the following to portal_properties / pipbox_properties

Album view <a> click handler:

```
{type:'overlay', subtype:'image', selector:'.photoAlbumEntry a', urlmatch:'/view$',
↪urlreplace:'/image_large'}
```

Note: portal_javascript must be in debug mode while testing different Products.PipBox handlers.

Upgrade tips

Description

Advanced tips for upgrading Plone.

Some of the information on this page is for Plone 4, which used the Archetypes content type framework. Plone 5.x uses the Dexterity content type framework.

General Tips

This guide contains some tips for Plone upgrades. For more information, see also the [official Plone upgrade guide](#)

Recommended setup

- Test the upgrade on your local development computer first.
- Create two buildouts: one for the old Plone version (your existing buildout) and one for the new version.

- Prepare the migration in the old buildout. After all preparations are done, copy the Data.fs and blobstorage to the new buildout and run `plone_migration` tool there.

Fix persistent utilities

You might need to clean up some leftovers from uninstalled add-ons which have not cleanly uninstalled.

Use this utility:

- <https://pypi.python.org/pypi/wildcard.fixpersistentutilities>

Note: Perform this against old buildout

Content Upgrades

For content migrations, `Products.contentmigration` can help you. Documentation on how to use it can be found on plone.org.

Migration from non-folderish to folderish Archetypes based content types

Non-folderish content types are missing some BTree attributes, which folderish content types have (See `Products.BTreeFolder2.BTreeFolder2Base._initBtrees`).

`plone.app.folder` provides an upgrade view to migrate pre-`plone.app.folder` (or non-folderish) types to the new BTree based implementation (defined in: `plone.app.folder.migration.BTreeMigrationView`).

To upgrade your non-folderish content types to folderish ones, just call `@@migrate-btrees` on your Plone site root, and you're done.

This applies to Archetypes based content types.

Upgrading theme

Make sure that your site them works on Plone 4. Official upgrade guide has tips how the theme codebase should be upgraded.

Theme fixing and `portal_skins`

Your theme might be messed up after upgrade.

Try playing around setting in *portal_skins Properties* tab. You can enable, disable and reorder skins layer applied in the theme.

Upgrade may change the default theme and you might want to restore custom theme in *portal_skins*.

Upgrade tips for `plone.app.discussion`

Enabling plone.app.discussion after Plone 4.1 upgrade

After migration from an earlier version of Plone, you will may notice that you do not have a *Discussion* control panel for `plone.app.discussion`, the new commenting infrastructure which now ships as part of new Plone installs beyond version 4.1. If a check of your *Site Setup* page reveals that you do not have the *Discussion* control panel, implement the following.

Install plone.app.discussion manually

1. Log into your Plone site as a user with Manager access
2. Browse to the following URL to manually install `plone.app.discussion`:

```
http://<your-plone-url>:<port>/<plone-instance>/portal_setup/manage_importSteps
```

3. In the *Select Profile or Snapshot* drop-down menu, select `Plone Discussions`.
4. Click the `Import all steps` button at the bottom of the page.
5. Confirm that *Discussion* is now present as a control panel in your *Site Setup*

Migrate existing comments

Follow the instructions regarding [How to migrate comments to plone.app.discussion](#) to migrate existing Plone comments.

Fixing Creator details on existing comments

You may notice that some of your site's comments have the user's ID as their *Creator* property. At time of writing (for `plone.app.discussion==2.0.10`), the *Creator* field should refer to the user's full name and not their user ID. You'll likely notice that a number of other fields, including `author_username`, `author_name` and `author_email` are not present on some of your migrated comments. Reasons why comments get migrated but unsuccessfully are being investigated.

This may change for future versions of `plone.app.discussion`. For now, though, having the user ID left as the *Creator* is less than helpful and means aspects like the username, name, and email not present affect usability of comments.

If a site has many comments with this issue, it is possible to step through all of them and correct them. Using a script like the following will process each of the affected comments accordingly:

```
from Products.CMFPlone.utils import getToolByName
from zope.app.component import hooks
from plone import api

context = hooks.getSite()

catalog = api.portal.get_tool(name='portal_catalog')
mtool = api.portal.get_tool(name='portal_membership')

brains = catalog.searchResults(object_provides='plone.app.discussion.interfaces.
↪ IComment')
for brain in brains:
    member = api.user.get(username=brain.Creator')
    comment = brain.getObject()
```

```
if member and not comment.author_username and not comment.author_name and not_
↳comment.author_email:
    fullname = member.getProperty('fullname')
    email = member.getProperty('email')
    if fullname and email:
        comment.author_username = brain.Creator #our borked user ID
        comment.creator = fullname
        comment.author_name = fullname
        comment.author_email = email
        comment.reindexObject()
        print 'Fixed and reindexed %s' % comment
    else:
        print 'Could not find properties for author of %s' % comment
```

This can be run anywhere an Acquisition context object is available, such as running your Zope instance in debug mode, an ipython prompt, or some other function on the filesystem. The `getSite()` function call can (and may need to) be replaced with some other pre-existing context object if that is more suitable.

Keep in mind that this script was successfully used in a situation where no possible collisions existed between correctly-migrated comments Creators' full names and user IDs (the code looks up the Creator in the hope of finding a valid Plone member). If you had a situation where you had some correctly migrated comments written by a user with ID `david` and full name of `Administrator`, and also had a user with the ID of `Administrator`, then this script may not be suitable. In the test situation, the three attributes of `author_username`, `author_name`, and `author_email` were observed as all being `None`, so in checking for this too, this may avoid problems. Test the code first with something like a `print` statement to ensure all comments will get modified correctly.

HTML manipulation and transformations

Description

How to programmatically rewrite HTML in Plone.

Introduction

It is recommended to use the `lxml` library for all HTML DOM manipulation in Python.

Plone is no exception.

Converting HTML to plain text

The most common use case is to override `SearchableText()` to return HTML content for `portal_catalog` for indexing.

- <http://stackoverflow.com/questions/6956326/custom-searchabletext-and-html-fields-in-plone>

Converting plain text to HTML

You can use `portal_transforms` to do plain text -> HTML conversion.

Below is an example how to create a Description field rendered with new line support.

Register the view in `configure.zcml`:

```
<configure
  xmlns="http://namespaces.zope.org/zope"
  xmlns:browser="http://namespaces.zope.org/browser"
  xmlns:plone="http://namespaces.plone.org/plone"
  i18n_domain="example.dexterityforms">

  ...

  <browser:page
    name="description-helper"
    for="*"
    class=".description.DescriptionHelper"
    permission="zope2.View"
  />

</configure>
```

Create a file `description.py` and add the following code:

```
from plone import api
from zope.interface import Interface
from Products.Five.browser import BrowserView

class DescriptionHelper(BrowserView):
    """
    A helper view which exports dublin core description w/new line support
    allowing several paragraphs in Plone's description field.
    """

    def render(self):
        """
        Get a content item description w/new line support.

        Transform hard lines to breaks in HTML.
        """

        # Call archetypes accessor
        text = self.context.Description()

        # Transform plain text description with ASCII newlines
        # to one with
        portal_transforms = api.portal.get_tool(name='portal_transforms')

        # Output here is a single <p> which contains <br /> for newline
        data = portal_transforms.convertTo('text/html', text, mimetype='text/-x-web-
↪intelligent')
        html = data.getData()
        return html
```

Now you can do in your page template

```
<metal:main-macro define-macro="main">

  <div tal:replace="structure provider:plone.abovecontenttitle" />

  <h1 metal:use-macro="here/kss_generic_macros/macros/generic_title_view">
```

```
    Title or id
</h1>

<div tal:replace="structure provider:plone.belowcontenttitle" />

<div class="documentDescription">
    <tal:desc replace="structure context/@@description-helper" />
</div>

...
```

More info

- <https://github.com/plone/plone.intelligenttext/tree/master/plone/intelligenttext>

Rewriting relative links

Below is an example which:

- rewrites all relative links of Page content as absolute;
- removes some nasty tags from Page content;
- outputs the folder content and subcontent as one continuous page;

This is suitable for e.g. printing the whole folder in one pass.

Register the view in `configure.zcml`:

```
<configure
    xmlns="http://namespaces.zope.org/zope"
    xmlns:browser="http://namespaces.zope.org/browser"
    >

    <browser:page
        for="Products.CMFCore.interfaces.IFolderish"
        name="help"
        permission="zope2.View"
        class=".help.Help"
        />

</configure>
```

Add the file `help.py`:

```
from lxml import etree
from StringIO import StringIO
import urlparse
from lxml import html

import zope.interface
from Products.Five.browser import BrowserView

def fix_links(content, absolute_prefix):
    """
    Rewrite relative links to be absolute links based on certain URL.

    @param html: HTML snippet as a string
```

```

"""

if type(content) == str:
    content = content.decode("utf-8")

parser = etree.HTMLParser()

content = content.strip()

tree = html.fragment_fromstring(content, create_parent=True)

def join(base, url):
    """
    Join relative URL
    """
    if not (url.startswith("/") or "://" in url):
        return urlparse.urljoin(base, url)
    else:
        # Already absolute
        return url

for node in tree.xpath('//*[@src]'):
    url = node.get('src')
    url = join(absolute_prefix, url)
    node.set('src', url)
for node in tree.xpath('//*[@href]'):
    href = node.get('href')
    url = join(absolute_prefix, href)
    node.set('href', url)

data = etree.tostring(tree, pretty_print=False, encoding="utf-8")

return data

def remove_bad_tags(content):
    """ Filter out HTML nodes which would prevent continuous printing """

    if type(content) == str:
        content = content.decode("utf-8")

    tree = html.fragment_fromstring(content, create_parent=True)

    # Title tag in the middle of page causes Firefox to choke and
    # aborts page rendering
    for node in tree.xpath('//title'):
        node.getparent().remove(node)

    data = etree.tostring(tree, pretty_print=False, encoding="utf-8")

    return data

class Help(BrowserView):
    """ Render all folder pages and subpages as continuous printable document """

    def update(self):

```

```
objects = []
# Walk through all objects recursively

def walk(folder, level):

    for id, object in folder.contentItems():

        if object.portal_type == "Image":
            continue

        # Output pages which have text payload
        if hasattr(object, "getText"):
            text = object.getText()
        else:
            text = ""

        objects.append({
            "object":object,
            "level":level,
            # We need to re-map relative links or
            # they are incorrect in rendered HTML output
            "text" : remove_bad_tags(fix_links(text, object.absolute_url()))
        })

        if object.portal_type == "Folder":
            walk(object, level+1)

walk(self.context, 1)

self.objects = objects
```

Add the `help.pt` template:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:tal="http://xml.zope.org/namespaces/tal"
      xmlns:metal="http://xml.zope.org/namespaces/metal"
      xmlns:i18n="http://xml.zope.org/namespaces/i18n"
      metal:use-macro="context/main_template/macros/master">
<body>

<metal:slot metal:fill-slot="content-title" i18n:domain="cmf_default">
  <h1>Site help</h1>

  <p class="discreet">
    Printable versions
  </p>
</metal:slot>

<metal:block fill-slot="top_slot" tal:define="dummy python:request.set('disable_border
→', 1)" />

<metal:slot metal:fill-slot="content-core" i18n:domain="cmf_default">

  <div class="help-all">
    <tal:rep repeat="page view/objects">
      <tal:def define="body page/text|nothing;title page/object/Title;level_
→page/level">
```

```
<div tal:condition="python:level==1" style="page-break-before:always">
<!-- --></div>
<h1 tal:condition="python:level==1" tal:content="title" />
<h2 tal:condition="python:level==2" tal:content="title" />
<h3 tal:condition="python:level>2" tal:content="title" />

<div class="help-body">
  <tal:body tal:replace="structure body" />
</div>

<div style="clear: both"><!-- --></div>

</tal:def>
</tal:rep>
</div>
</metal:slot>
</body>
</html>
```

SQL

Description

Using SQL databases (MySQL, PostgreSQL, others) in Plone

Introduction

If you are building the codebase Plone behaves as any other Python application.

- Write your SQL related code using known available Python SQL libraries and frameworks
- Plug your code to Plone HTML pages through *views*

Example Python SQL libraries

- <http://www.sqlalchemy.org/>

ZSQL

ZSQL is something probably written before you knew what SQL is. Never ever use ZSQL in new code. It's not following any modern best practices and has history of 1990s code. You have been warned. Stay away. The grue is near.

Changing Portal Transforms Settings via Python

Introduction

If you have to change some portal_transforms settings you can't use a Generic Setup config file for this. But you can change it with python and a Generic Setup import step.

Warning: Security: The configuration shown below allows users to use nasty HTML tags which can be a security issue if not used carefully.

Let's say we have a Plone package called MY.PACKAGE.

Writing an Generic Setup Import Step Method

This setup method is defined in MY.PACKAGE/setuphandlers.py. It configures the safe_html portal_transform a bit less paranoid about nasty tags and valid_tags, so that content managers are allowed to insert iframe, object, embed, param, script, style, tags and more into the TinyMCE editor:

```
import logging
from plone import api
from Products.PortalTransforms.Transform import make_config_persistent

logger = logging.getLogger('MY.PACKAGE.setuphandlers')

def isNotThisProfile(context, marker_file):
    return context.readDataFile(marker_file) is None

def setup_portal_transforms(context):
    if isNotThisProfile(context, 'MY.PACKAGE-PROFILENAME.txt'): return

    logger.info('Updating portal_transform safe_html settings')

    tid = 'safe_html'

    pt = api.portal.get_tool(name='portal_transforms')
    if not tid in pt.objectIds(): return

    trans = pt[tid]

    tconfig = trans._config
    tconfig['class_blacklist'] = []
    tconfig['nasty_tags'] = {'meta': '1'}
    tconfig['remove_javascript'] = 0
    tconfig['stripped_attributes'] = ['lang', 'valign', 'halign', 'border',
                                     'frame', 'rules', 'cellspacing',
                                     'cellpadding', 'bgcolor']
    tconfig['stripped_combinations'] = {}
    tconfig['style_whitelist'] = ['text-align', 'list-style-type', 'float',
                                  'width', 'height', 'padding-left',
                                  'padding-right'] # allow specific styles for
                                                    # TinyMCE editing

    tconfig['valid_tags'] = {
        'code': '1', 'meter': '1', 'tbody': '1', 'style': '1', 'img': '0',
        'title': '1', 'tt': '1', 'tr': '1', 'param': '1', 'li': '1',
        'source': '1', 'tfoot': '1', 'th': '1', 'td': '1', 'dl': '1',
        'blockquote': '1', 'big': '1', 'dd': '1', 'kbd': '1', 'dt': '1',
        'p': '1', 'small': '1', 'output': '1', 'div': '1', 'em': '1',
        'datalist': '1', 'hgroup': '1', 'video': '1', 'rt': '1', 'canvas': '1',
        'rp': '1', 'sub': '1', 'bdo': '1', 'sup': '1', 'progress': '1',
        'body': '1', 'acronym': '1', 'base': '0', 'br': '0', 'address': '1',
        'article': '1', 'strong': '1', 'ol': '1', 'script': '1', 'caption': '1',
        'dialog': '1', 'col': '1', 'h2': '1', 'h3': '1', 'h1': '1', 'h6': '1',
```



```

'h4': '1', 'h5': '1', 'header': '1', 'table': '1', 'span': '1',
'area': '0', 'mark': '1', 'dfn': '1', 'var': '1', 'cite': '1',
'thead': '1', 'thead': '1', 'hr': '0', 'link': '1', 'ruby': '1',
'b': '1', 'colgroup': '1', 'keygen': '1', 'ul': '1', 'del': '1',
'iframe': '1', 'embed': '1', 'pre': '1', 'figure': '1', 'ins': '1',
'aside': '1', 'html': '1', 'nav': '1', 'details': '1', 'u': '1',
'samp': '1', 'map': '1', 'object': '1', 'a': '1', 'footer': '1',
'i': '1', 'q': '1', 'command': '1', 'time': '1', 'audio': '1',
'section': '1', 'abbr': '1'}
make_config_persistent(tconfig)
trans.__p_changed = True
trans.reload()

```

Registering the Import Step Method with Generic Setup

Add an import step in MY.PACKAGE/MYPROFILESDIR/PROFILENAME/import_steps.xml like so::

```

<?xml version="1.0"?>
<import-steps>
  <import-step
    id="MY.PACKAGE-portal_transforms"
    handler="MY.PACKAGE.setuphandlers.setup_portal_transforms"
    title="MY.PACKAGE portal_transforms setup"
    version="1.0">
    <dependency step="plone-final"/>
  </import-step>
</import-steps>

```

Create the File MY.PACKAGE/MYPROFILESDIR/PROFILENAME/MY.PACKAGE-PROFILENAME.txt, so that this import step is not run for any profile but just for this one.

Calling the Import Step Method in Management Interface, portal_setup

Go to your site's portal_setup in Management Interface, select your registered profile and import the import step "MY.PACKAGE portal_transforms setup".

Looking ahead towards Plone 5

Concerns regarding removal of portal_skins and reliance on browser views

Specific Things We Like to Do with portal_skins

This document includes a bunch of specific use cases showing how we as integrators typically rely on portal_skin.

Nathan Van Gheem's responses below are indented.

Live Sites

We can modify live sites' appearance without having to touch the file system by putting things in the custom folder.

Plone has, and will always try to provide a rich TTW editing and customization story. This is true with plone.app.theming and diazo. If all skins are removed, we WILL provide an alternative way to customize template TTW. Right now, it looks to me like making portal_view_customizations work better.

No Filesystem or Buildout Access

We often do not have access to the file system nor can we run buildout.

See Live Sites response.

Customizing a collection's display

We have some custom content types that we want to display using a collection. We build the collection and specify “item type”. We want the display to show fields that are unique to the custom content types. We locate the collection view template, customize it, rename it (to, say, custom_collection_view), enhance it to show the additional field values, then in portal_types we add the new custom_collection_view to the list of available views for Topics. The collection’s “Display” menu now includes the new custom_collection_view.

First off, best case we still have a story to do the exact same thing only with portal_view_customizations.

Secondly, it can be easier to hit that use-case with a combination of collective.listingviews and diazo. There has been discussion of integrating a lot of what collective.listingviews does and more into plone.app.theming.

Creating a cloned content type so that it has a different default view

Let’s say a site has a custom content type based on Document but we want to have the default view include boilerplate text around the rich text and description. We would go to portal_types, clone the Document type, rename the cloned type “Project”. Then we go to portal_skins, find document_view, customize it, rename it to project_view, and add the boilerplate text we want. Then back in portal_types for Project, we change the default view to project_view. This way, anywhere in the site we create a Project object, its default view (its only view) shows the boilerplate text we wanted.

Cloned content types will still be available with dexterity. In fact, it’ll be incredibly more robust and powerful.

For the views, look to the previous point about using collective.listingviews and diazo.

Classic Portlets

We use classic portlets a lot to put together (quickly) something that displays arbitrary content.

There is nothing scheduled to get rid of portlet or the classic portlet right now. portal_skins will still be there.

That being said, I might need more specific use-cases of how you’re using classic portlets in order to explain how it’d be a replacement.

Things We Don’t Like About Having to Rely Only on Browser Views

Why browser views are hard for integrators (non-developers):

- We may not have file system access

- We may not want to have to (and are not in fact able to) create a product to register a new view
- We may not want to have to re-run buildout (nor are we able to) to register a new view
- Unless a browser view is correctly registered, customizing it via `portal_view_customizations` breaks Python methods associated with the view

I hope I've addressed your concerns. The final point is valid and a concern of mine also. We'll need to make sure there is a way to customize all templates safely. I sort of hope people simply won't be doing TTW customizations of templates as much anymore though and they'll just use diazo with something like listingviews.

Others might have different ideas about how things will work. Dylan Jay might be someone that can give really good answers regarding these questions.

A good discussion regarding some of these issues can be found at: <http://plone.293351.n2.nabble.com/enhanced-collections-views-td7565206.html;cid=1372262563684-127>

The final response there has a good overview.

Debugging Plone

This section contains tips how to debug your code.

Logging

Description

How to write log output from your Plone add-on program code.

Introduction

Python `logging` package is used to log from Plone.

Viewing Logs In Real Time

The best way to trace log messages when developing, is start the Zope instance in foreground mode. Log messages are printed to the console (stdout).

You can of course also view the logs from the logfile

```
tail -f var/log/instance.log
```

Press CTRL+C to abort.

The Site Error Log Service

Plone sites contain error log service which is located as `error_log` in the site root. It logs site exceptions and makes the tracebacks accessible from Plone control panel and the Management Interface.

The service is somewhat archaic and can log exceptions only, not plain error messages.

Log Level

Default log level is INFO. To enable more verbose logging, edit `buildout.cfg`,

Change log level by editing `[instance]` section `event-log-level`

```
[instance]
event-log-level = debug
```

More information

- <https://pypi.python.org/pypi/plone.recipe.zope2instance>

Logging From Python Code

Example

```
import logging

logger = logging.getLogger("Plone")

class MySomething(object):
    ...
    def function(self):
        logger.info("Reached function()")
    ...
```

Logging From Page Templates And RestrictedPython Scripts

Python logging module doesn't provide Zope 2 security assertions and does not work in *RestrictedPython Python scripts*.

However, you can use `context.plone_log()` method logging in the sandboxed execution mode.

Example:

```
context.plone_log("This is so fun")
```

Forcing Log Level And Output

The following snippet forces the log level of Python logging for the duration of the process by modifying the root logger object:

```
# Force application logging level to DEBUG and log output to stdout for all loggers
import sys, logging

root_logger = logging.getLogger()
root_logger.setLevel(logging.DEBUG)

handler = logging.StreamHandler(sys.stdout)
formatter = logging.Formatter("%(asctime)s - %(name)s - %(levelname)s - %(message)s")
handler.setFormatter(formatter)
root_logger.addHandler(handler)
```

Temporarily Capturing Log Output

You can capture Python logging output temporarily to a string buffer. This is useful if you want to use logging module to record the status of long running operations and later show to the end user, who does not have access to file system logs, how the operation proceeded.

Below is an *BrowserView* code example.

Example view code:

```
import logging
from StringIO import StringIO

from Products.Five import BrowserView

from xxx.objects.interfaces import IXXXResearcher
from Products.statusmessages.interfaces import IStatusMessage

from xxx.objects.sync import sync_with_xxx

logger = logging.getLogger("XXX sync")

class SyncAll(BrowserView):
    """
    Update all researcher data on the site from XXX (admin action)
    """

    def sync(self):
        """
        Search all objects of certain type on the site and
        sync them with a remote site.
        """

        brains = self.context.portal_catalog(object_provides=IXXXResearcher.__
→ identifier__)
        for brain in brains:
            object = brain.getObject()
            sync_with_xxx(object, force=True)

    def startCapture(self, newLogLevel = None):
        """ Start capturing log output to a string buffer.

        http://docs.python.org/release/2.6/library/logging.html

        @param newLogLevel: Optionally change the global logging level, e.g. logging.
→ DEBUG
        """
        self.buffer = StringIO()

        print >> self.buffer, "Log output"

        rootLogger = logging.getLogger()

        if newLogLevel:
            self.oldLogLevel = rootLogger.getEffectiveLevel()
            rootLogger.setLevel(newLogLevel)
        else:
            self.oldLogLevel = None
```

```
self.logHandler = logging.StreamHandler(self.buffer)
formatter = logging.Formatter("%(asctime)s - %(name)s - %(levelname)s -
↪ %(message)s")
self.logHandler.setFormatter(formatter)
rootLogger.addHandler(self.logHandler)

def stopCapture(self):
    """ Stop capturing log output.

    @return: Collected log output as string
    """

    # Remove our handler
    rootLogger = logging.getLogger()

    # Restore logging level (if any)
    if self.oldLogLevel:
        rootLogger.setLevel(self.oldLogLevel)

    rootLogger.removeHandler(self.logHandler)

    self.logHandler.flush()
    self.buffer.flush()

    return self.buffer.getvalue()

def __call__(self):
    """ Process the form.

    Process the form, log the output and show the output to the user.
    """

    self.logs = None

    if "sync-now" in self.request.form:
        # Form button was pressed

        # Open Plone status messages interface for this request
        messages = IStatusMessage(self.request)

        try:
            self.startCapture(logging.DEBUG)

            logger.info("Starting full site synchronization")

            # Do the long running,
            # lots of logging stuff
            self.sync()

            logger.info("Successfully done")

            # It worked! Trolololo.
            messages.addStatusMessage("Sync done")

        except Exception, e:
            # Show friendly error message
```

```

        logger.exception(e)
        messages.addStatusMessage(u"It did not work out:" + unicode(e))

    finally:
        # Put log output for the page template access
        self.logs = self.stopCapture()
    return self.index()

```

The related page template

```

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
      lang="en"
      metal:use-macro="here/main_template/macros/master"
      i18n:domain="xxx.objects">
<body>
  <div metal:fill-slot="main">
    <tal:main-macro metal:define-macro="main">

      <h1 class="documentFirstHeading">
        XXX site update
      </h1>

      <p class="documentDescription">
        Update all researches from XXX
      </p>

      <div tal:condition="view/logs">
        <p>Sync results:</p>
        <pre tal:content="view/logs" />
      </div>

      <form action="@@syncall" method="POST">
        <button type="submit" name="sync-now">
          Sync now
        </button>
      </form>

    </tal:main-macro>
  </div>
</body>
</html>

```

Registering the view in ZCML:

```

<browser:view
    for="Products.CMFPlone.interfaces.IPloneSiteRoot"
    name="syncall"
    class=".views.SyncAll"
    permission="cmf.ManagePortal"
/>

```

transaction_note()

Leave a note on Zope's *History* tab.

- <https://github.com/plone/Products.CMFPlone/blob/master/Products/CMFPlone/utils.py#L382>

Python debugging

Description

Using Python command-line debugger (pdb) to debug Plone and Python applications.

Introduction

The Python debugger (pdb) is an interactive command-line debugger.

It is very limited in functionality, but it will work in every environment and type of console. Plone also has through-the-web-browser PBD debugging add-on products.

Note: pdb is not the same as the Python interactive shell. pdb allows you to step through the code, whilst the Python shell allows you just to inspect and manipulate objects.

If you wish to play around with Zope in interactive Python shell or run scripts instead of debugging (exceptions), please read *Command line* documentation.

See also

Using pdb

Go to your code and insert the statement `import pdb; pdb.set_trace()` at the point where you want have a closer look. Next time the code is run, the execution will stop there and you can examine the current context variables from a Python command prompt.

After you have added `import pdb; pdb.set_trace()` to your code, stop Zope and start it in the foreground using the `bin/instance fg` command.

TextMate support for pdb can be found at <https://pypi.python.org/pypi/PdbTextMateSupport/0.3>.

`mr.freeze` allows traces to be added without restarting: <https://pypi.python.org/pypi/mr.freeze>.

Example:

```
class AREditForm(crud.EditForm):
    """ Present edit table containing rows per each item added and delete controls """
    editsubform_factory = AREditSubForm

    template = viewpagetemplatefile.ViewPageTemplateFile('ar-crud-table.pt')

    @property
    def fields(self):

        #
        # Execution will stop here and interactive Python prompt is opened
        #

        import pdb ; pdb.set_trace()
        constructor = ARFormConstructor(self.context, self.context.context, self.
↪request)
        return constructor.getFields()
```


Pretty printing objects

Example:

```
>>> pp folder.__dict__
{
  '_Access_contents_information_Permission': ['Anonymous',
                                              'Manager',
                                              'Reviewer'],
  '_List_folder_contents_Permission': ('Manager', 'Owner', 'Member'),
  '_Modify_portal_content_Permission': ('Manager', 'Owner'),
  '_View_Permission': ['Anonymous', 'Manager', 'Reviewer'],
  '__ac_local_roles__': {'gregweb': ['Owner']},
  '_objects': ({'meta_type': 'Document', 'id': 'doc1'},
               {'meta_type': 'Document', 'id': 'doc2'}),
  'contributors': (),
  'creation_date': DateTime('2005/02/14 20:03:37.171 GMT+1'),
  'description': 'Dies ist der Mitglieder-Ordner.',
  'doc1': <Document at doc1>,
  'doc2': <Document at doc2>,
  'effective_date': None,
  'expiration_date': None,
  'format': 'text/html',
  'id': 'folder',
  'language': '',
  'modification_date': DateTime('2005/02/14 20:03:37.203 GMT+1'),
  'portal_type': 'Folder',
  'rights': '',
  'subject': (),
  'title': "Documents",
  'workflow_history': {'folder_workflow': ({'action': None,
      'review_state': 'visible', 'comments': '', 'actor': 'gregweb',
      'time': DateTime('2005/02/14 20:03:37.187 GMT+1')},)}
```

Useful pdb commands

Just type the command and hit enter.

s step into, go into the function in the cursor

n step over, execute the function under the cursor without stepping into it

c continue, resume program

w where am I? displays current location in stack trace

b set breakpoint

cl clear breakpoint

bt print stack trace

up go to the scope of the caller function

pp pretty print object

until Continue execution until the line with the line number greater than the current one is reached or when returning from current frame

Note: The `until` command (or `unt`) is available only on Plone 4.x or superior as it is a new feature provided by the `pdb` module under Python 2.6.

Useful `pdb` snippets

Output object's class:

```
(Pdb) print obj.__class__
```

Output object attributes and methods:

```
(Pdb) for i in dir(obj): print i
```

Print local variables in the current function:

```
(Pdb) print locals()
```

Dumping incoming HTTP GET or HTTP POST:

```
(Pdb) print "Got request:"  
(Pdb) for i in self.request.form.items(): print i
```

Executing code on the context of the current stack frame:

```
(Pdb) pp my_tags  
['bar', 'barbar']  
  
(Pdb) !my_tags = ['foo', 'foobar']  
(Pdb) pp my_tags  
['foo', 'foobar']
```

Note: The example above will modify the previous value of the variable `my_tags` in the current stack frame.

Automatically start debugger when exception is raised (browser)

You can start interactive through-the-browser Python debugger when your site throws an exception.

Instead of getting “We’re sorry there seems to be an error...” page you get a `pdb` prompt which allows you to debug the exception. This is also known as post-mortem debugging.

This can be achieved with ‘*Products.PDBDebugMode*’ add-on.

- <https://pypi.python.org/pypi/Products.PDBDebugMode>

Note: `PDBDebugMode` is not safe to install on the production server due to sandbox security escape.

Automatically start debugger when exception is raised (command line)

Note: This cannot be directly applied to a web server, but works with command line scripts.

Note: This does not work with Zope web server launch as it forks a process.

Example:

```
python -m pdb myscript.py
```

Hit `c` and `enter` to start the application. It keeps running, until an uncaught exception is raised. At this point, it falls back to the `pdb` debug prompt.

For more information see

- <http://docs.python.org/library/pdb.html>

Testing Plone

This section contains tips how to test your code.

Unit testing

Introduction

Unit tests are automated tests created by the developer to ensure that the add-on product is intact in the current product configuration. Unit tests are regression tests and are designed to catch broken functionality over the code evolution.

Running unit tests

Since Plone 4, it is recommended to use `zc.testrunner` to run the test suites. You need to add it to your `buildout.cfg`, so that the `test` command will be generated.

```
parts =
    ...
    test

[test]
recipe = zc.recipe.testrunner
defaults = ['--auto-color', '--auto-progress']
eggs =
    ${instance:eggs}
```

Running tests for one package:

```
bin/test -s package.subpackage
```

Running tests for one test case:

```
bin/test -s package.subpackage -t TestCaseClassName
```

Running tests for two test cases:

```
bin/test -s package.subpackage -t TestClass1|TestClass2
```

To drop into the pdb debugger after each test failure:

```
bin/test -s package.subpackage -D
```

To exclude tests:

```
bin/test -s package.subpackage -t !test_name
```

To list tests that will be run:

```
bin/test -s package.subpackage --list-tests
```

The following will run tests for *all* Plone add-ons: useful to check whether you have a set of component that function well together:

```
bin/test
```

Warning: The test runner does not give an error if you supply invalid package and test case name. Instead it just simply doesn't execute tests.

More information:

- <https://plone.org/documentation/manual/upgrade-guide/version/upgrading-plone-3-x-to-4.0/updating-add-on-products-for-plone-4.0/no-longer-bin-instance-test-use-zc.recipe.testrunner>

AttributeError: 'module' object has no attribute 'test_suite'

If you get the above error message there are two potential reasons:

- You have both a `tests.py` file and a `tests` folder.
- Old version: Zope version X unit test framework was updated not to need an explicit `test_suite` declaration in the `test` module any more. Instead, all subclasses of `TestCase` are automatically picked. However, this change is backwards incompatible.

Test coverage

Zope test running can show how much of your code is covered by automatic tests:

- <https://pypi.python.org/pypi/plone.testing#coverage-reporting>

Running tests against Python egg

You might need to add additional `setup.py` options to get your tests work

- <http://rpatterson.net/blog/running-tests-in-egg-buildouts>

Creating unit tests

For any new test suites, you should be using `plone.app.testing`, your next step should be to *read the documentation* `</external/plone.app.testing/docs/source>`.

You may come across `Products.PloneTestCase` `<https://pypi.python.org/pypi/Products.PloneTestCase>` in older code. Also interesting is `ZopeTestCase` `<http://www.zope.org/Members/shh/ZopeTestCaseWiki/ApiReference>`.

Miscellaneous hints

Setting log level in unit tests

Many components use the `DEBUG` output level, while the default output level for unit testing is `INFO`. Import messages may go unnoticed during the unit test development.

Add this to your unit test code:

```
def enableDebugLog(self):
    """ Enable context.plone_log() output from Python scripts """
    import sys, logging
    from Products.CMFPlone.log import logger
    logger.root.setLevel(logging.DEBUG)
    logger.root.addHandler(logging.StreamHandler(sys.stdout))
```

Test outgoing email messages

The `MailHost` code has changed in Plone 4. For more detail about the changes please read the relevant section in the [Plone Upgrade Guide](#). According to that guide we can reuse some of the test code in `Products.CMFPlone.tests`.

Here's some example of a `unittest.TestCase` based on the excellent `plone.app.testing` framework. Adapt it to your own needs.

```
#Pythonic libraries
import unittest2 as unittest
from email import message_from_string

#Plone
from plone.app.testing import TEST_USER_NAME, TEST_USER_ID
from plone.app.testing import login, logout
from plone.app.testing import setRoles
from plone.testing.z2 import Browser

from Acquisition import aq_base
from zope.component import getSiteManager
from Products.CMFPlone.tests.utils import MockMailHost
from Products.MailHost.interfaces import IMailHost
import transaction

#hkl namespace
from holokinesislubros.purchaseorder.testing import \
    HKL_PURCHASEORDER_FUNCTIONAL_TESTING

class TestOrder(unittest.TestCase):
```

```

layer = HKL_PURCHASEORDER_FUNCTIONAL_TESTING

def setUp(self):
    self.app = self.layer['app']
    self.portal = self.layer['portal']
    self.portal._original_MailHost = self.portal.MailHost
    self.portal.MailHost = mailhost = MockMailHost('MailHost')
    sm = getSiteManager(context=self.portal)
    sm.unregisterUtility(provided=IMailHost)
    sm.registerUtility(mailhost, provided=IMailHost)

    self.portal.email_from_address = 'noreply@holokinesislibros.com'
    transaction.commit()

def tearDown(self):
    self.portal.MailHost = self.portal._original_MailHost
    sm = getSiteManager(context=self.portal)
    sm.unregisterUtility(provided=IMailHost)
    sm.registerUtility(aq_base(self.portal._original_MailHost),
                      provided=IMailHost)

def test_mockmailhost_setting(self):
    #open contact form
    browser = Browser(self.app)
    browser.open('http://nohost/plone/contact-info')
    # Now fill in the form:

    form = browser.getForm(name='feedback_form')
    form.getControl(name='sender_fullname').value = 'T\xc3\xa4st user'
    form.getControl(name='sender_from_address').value = 'test@plone.test'
    form.getControl(name='subject').value = 'Saluton amiko to\xc3\xb1o'
    form.getControl(name='message').value = 'Message with funny chars:
→ \xc3\xa1\xc3\xa9\xc3\xad\xc3\xb3\xc3\xba\xc3\xb1.'

    # And submit it:
    form.submit()
    self.assertEqual(browser.url, 'http://nohost/plone/contact-info')
    self.assertIn('Mail sent', browser.contents)

    # As part of our test setup, we replaced the original MailHost with our
    # own version. Our version doesn't mail messages, it just collects them
    # in a list called ``messages``:
    mailhost = self.portal.MailHost
    self.assertEqual(len(mailhost.messages), 1)
    msg = message_from_string(mailhost.messages[0])

    self.assertEqual(msg['MIME-Version'], '1.0')
    self.assertEqual(msg['Content-Type'], 'text/plain; charset="utf-8"')
    self.assertEqual(msg['Content-Transfer-Encoding'], 'quoted-printable')
    self.assertEqual(msg['Subject'], '=?utf-8?q?Saluton_amiko_to=C3=B1o?='')
    self.assertEqual(msg['From'], 'noreply@holokinesislibros.com')
    self.assertEqual(msg['To'], 'noreply@holokinesislibros.com')
    msg_body = msg.get_payload()
    self.assertIn(u'Message with funny chars: =C3=A1=C3=A9=C3=AD=C3=B3=C3=BA=C3=B1
→ ',
                  msg_body)

```

Unit testing and the Zope component architecture

If you are dealing with the Zope component architecture at a low level in your unit tests, there are some things to remember, because the global site manager doesn't behave properly in unit tests.

See discussion: <http://plone.293351.n2.nabble.com/PTC-global-components-bug-tp3413057p3413057.html>

Below are examples how to run special ZCML snippets for your unit tests.

```
import unittest
from base import PaymentProcessorTestCase
from Products.Five import zcml
from zope.configuration.exceptions import ConfigurationError
from getpaid.paymentprocessors.registry import paymentProcessorRegistry

configure_zcml = '''
<configure
  xmlns="http://namespaces.zope.org/zope"
  xmlns:five="http://namespaces.zope.org/five"
  xmlns:paymentprocessors="http://namespaces.plonegetpaid.com/paymentprocessors"
  i18n_domain="foo">

  <paymentprocessors:registerProcessor
    name="dummy"
    processor="getpaid.paymentprocessors.tests.dummies.DummyProcessor"
    selection_view="getpaid.paymentprocessors.tests.dummies.DummyButton"
    thank_you_view="getpaid.paymentprocessors.tests.dummies.DummyThankYou"
  />

</configure>'''

bad_processor_zcml = '''
<configure
  xmlns="http://namespaces.zope.org/zope"
  xmlns:five="http://namespaces.zope.org/five"
  xmlns:paymentprocessors="http://namespaces.plonegetpaid.com/paymentprocessors"
  i18n_domain="foo">

  <paymentprocessors:registerProcessor
    name="dummy"
    selection_view="getpaid.paymentprocessors.tests.dummies.DummyButton"
    thank_you_view="getpaid.paymentprocessors.tests.dummies.DummyThankYou"
  />

</configure>'''

class TestZCML(PaymentProcessorTestCase):
    """ Test ZCML directives """

    def test_register(self):
        """ Check that ZCML entry gets added to our processor registry """
```

```
zcml.load_string(configure_zcml)

# See that our processor got registered
self.assertEqual(len(papaymentProcessorRegistry.items()), 1)

def test_bad_processor(self):
    """ Check that ZCML entry which has bad processor declaration is caught """

    try:
        zcml.load_string(bad_processor_zcml)
        raise AssertionError("Should not be never reached")
    except ConfigurationError, e:
        pass
```

Functional testing

Description

Functional testing tool allows you to use scripted browser to load pages from your site and fill in forms automatically.

Introduction

PloneTestCase product provides `FunctionalTestCase` base class for functional testing. Unlike unit tests, functional tests simulate real HTTP requests with transaction life cycle.

- Functional tests has different transaction for each `browser.open()` request
- Functional tests do traversing and can check e.g. for cookie based permissions
- Unit test method is executed in a single transaction and this might make impossible to test cache related behavior

Test browser

Plone uses `Products.Five.testbrowser` as a browser emulator used in functional tests. It is based on `zope.testbrowser` package. You can find more information in the [zope.testbrowser docs home page](#). The API is described in [zope.testbrowser.interfaces \(3.4 used by Plone 3\)](#).

Warning: There also exists old `zc.testbrowser`, which is a different package with similar name.

All code assumes here is executed in unit test context where `self.portal` is your unit test site instance.

Recording tests

You can record functional tests through the browser. Think it as a Microsoft Word macro recorder kind of thing.

- <http://pyyou.wordpress.com/2008/04/11/how-to-install-zopetestrecorder-with-buildout/>
- <https://pypi.python.org/pypi/zope.testrecorder>

Functional test skeleton

First see collective.testlayer package which does some of the things described below

- <https://pypi.python.org/pypi/collective.testcaselayer>

Example code:

```
from Products.Five.testbrowser import Browser
from Products.PloneTestCase import PloneTestCase as ptc

class BaseFunctionalTestCase(ptc.FunctionalTestCase):
    """ This is a base class for functional test cases for your custom product.
    """

    def afterSetUp(self):
        """
        Show errors in console by monkey patching site error_log service
        """

        ptc.FunctionalTestCase.afterSetUp(self)

        self.browser = Browser()
        self.browser.handleErrors = False # Don't get HTTP 500 pages

        self.portal.error_log._ignored_exceptions = ()

    def raising(self, info):
        import traceback
        traceback.print_tb(info[2])
        print info[1]

    from Products.SiteErrorLog.SiteErrorLog import SiteErrorLog
    SiteErrorLog.raising = raising

    def loginAsAdmin(self):
        """ Perform through-the-web login.

        Simulate going to the login form and logging in.

        We use username and password provided by PloneTestCase.

        This sets session cookie for testbrowser.
        """
        from Products.PloneTestCase.setup import portal_owner, default_password

        # Go admin
        browser = self.browser
        browser.open(self.portal.absolute_url() + "/login_form")
        browser.getControl(name='__ac_name').value = portal_owner
        browser.getControl(name='__ac_password').value = default_password
        browser.getControl(name='submit').click()
```

Preparing error logger

Since zope.testbrowser uses normal Plone paging mechanism, you won't get nice tracebacks to your console.

The following snippet allows you to extract traceback data from `site.error_log` utility and print it to the console. Put it to your `afterSetUp()`:

```
self.browser.handleErrors = False
self.portal.error_log._ignored_exceptions = ()

def raising(self, info):
    import traceback
    traceback.print_tb(info[2])
    print info[1]

from Products.SiteErrorLog.SiteErrorLog import SiteErrorLog
SiteErrorLog.raising = raising
```

Opening an URL

Example:

```
from Products.Five.testbrowser import Browser

self.browser = Browser()

self.browser.open(self.portal.absolute_url())
```

Logging in

Example:

```
from Products.PloneTestCase.setup import portal_owner, default_password

# Go admin
browser.open(self.portal.absolute_url() + "/login_form")
browser.getControl(name='__ac_name').value = portal_owner
browser.getControl(name='__ac_password').value = default_password
browser.getControl(name='submit').click()
```

Logout

Example:

```
def logoutWithTestBrowser(self):
    """
    """
    self.browser.open(self.portal.absolute_url() + '/logout')
    html = self.browser.contents
    self.assertTrue("You are now logged out" in html)
```

Showing the contents from the last request

After test browser has opened an URL its content can be read from `browser.contents` variable.

Example:

```
print browser.contents # browser is zope.testbrowser.Browser instance
```

Getting a form handler

You can use `testbrowser.getForm()` to access different forms on a page.

Form look-up is available by name or index.

Example:

```
form = browser.getForm(index=2) # Skip login and search form on Plone 4
```

Listing available form controls

You can do the following to know what content your form has eaten

- the mechanize browser instance that is used through `zope.testbrowser`. `zope.testbrowser` internally uses a test-browser provided by the mechanize package. The mechanize objects are saved in `browser.mech_browser` and as attributes on different other instances returned by `zope.testbrowser`. mechanize has a different, less convenient api, but also provides more options. To see a list of all controls in a for you can do e.g.:

```
# get the login form from the zope.testbrowser
login_form = self.browser.getForm('login_form')
# get and print all controls
controls = login_form.mech_form.controls
for control in controls:
    print "%s: %s" % (control.attrs['name'], control.attrs['type'])
```

... or one-liner ...:

```
for c in form.mech_form.controls: print c
```

- the HTML page source code:

```
print browser.contents
```

Filling in a text field on a page

You can manipulate value of various form input controls.

Example how to submit Plone search page:

```
self.browser.open(self.portal.absolute_url() + "/search")

# Input some values to the search that we see we get
# zero hits and at least one hit
for search_terms in [u"Plone", u"youcantfindthis"]:
    form = self.browser.getForm("searchform")

    # Fill in the search field
    input = form.getControl(name="SearchableText")
    input.value = search_terms

    # Submit the search form
    form.submit(u"Search")
```

Selecting a checkbox

Checkboxes are usually presented as name:list style names:

```
checkbox = form.getControl(name="myitem.select:list")
checkbox.value = [u"selected"]
```

Clicking a button

Example:

```
button = form.getControl(name="mybuttonname")
button.click()
```

If you have a form instance, you can use the submit action. To click on the Button labeled “Log in” in the login form, you do:

```
login_form = self.browser.getForm('login_form')
login_form.submit('Log in')
```

Checking Unauthorized response

Example:

```
def checkIsUnauthorized(self, url):
    """
    Check whether URL gives Unauthorized response.
    """

    import urllib2

    # Disable redirect on security error
    self.portal.acl_users.credentials_cookie_auth.login_path = ""

    # Unfuse exception tracking for debugging
    # as set up in afterSetUp()
    self.browser.handleErrors = True

    def raising(self, info):
        pass
    self.portal.error_log._ignored_exceptions = ("Unauthorized")
    from Products.SiteErrorLog.SiteErrorLog import SiteErrorLog
    SiteErrorLog.raising = raising

    try:
        self.browser.open(url)
        raise AssertionError("No Unauthorized risen:" + url)
    except urllib2.HTTPError, e:
        # Mechanize, the engine under testbrowser
        # uses urlllib2 and will raise this exception
        self.assertEqual(e.code, 401, "Got HTTP response code:" + str(e.code))
```

Another example where test browser / Zope 2 publisher where invalidly handling Unauthorized exception:

```
def test_anon_access_forum(self):
    """
    Anonymous users should not be able to open the forum page.
    """

    self.portal.error_log._ignored_exceptions = ()
    self.portal.acl_users.credentials_cookie_auth.login_path = ""

    exception = None
    try:
        self.browser.open(self.portal.intranet.forum.absolute_url())
    except:
        # Handle a broken case where
        # test browser spits out an exception without a base class (WTF)
        import sys
        exception = sys.exc_info()[0]

    self.assertFalse(exception is None)
```

Checking a HTTP response header

Exaple:

```
self.assertEqual(self.browser.headers["Content-type"], 'application/octet-stream')
```

Checking HTTP exception

Example how to check for HTTP 500 Internal Server Error:

```
def test_no_language(self):
    """ Check that language parameter is needed and nothing is executed unless it is_
    ↪given. """

    from urllib2 import HTTPError
    try:
        self.browser.handleErrors = True # Don't get HTTP 500 pages
        url = self.portal.absolute_url() + "/@@mobile_sitemap?mode=mobile"
        self.browser.open(url)
        # should cause HTTPError: HTTP Error 500: Internal Server Error
        raise AssertionError("Should be never reached")
    except HTTPError, e:
        pass
```

Setting test browser headers

Headers must be passed to underlying PublisherMechanizeBrowser instance and test browser must be constructed based on this instance.

Note: When passing parameters to PublisherMechanizeBrowser.addheaders HTTP prefix will be automatically added to header name.

Add header to browser

```
>>> from Products.Five.testbrowser import Browser
>>> browser = Browser()
>>> browser.addHeader(key, value)
```

Setting user agent

Example:

```
class BaseFunctionalTestCase(ptc.FunctionalTestCase):

    def setUA(self, user_agent):
        """
        Create zope.testbrowser Browser with a specific user agent.
        """

        # Be sure to use Products.Five.testbrowser here
        self.browser = UABrowser(user_agent)
        self.browser.handleErrors = False # Don't get HTTP 500 pages

from zope.testbrowser import browser
from Products.Five.testbrowser import PublisherHTTPHandler
from Products.Five.testbrowser import PublisherMechanizeBrowser

class UABrowser(browser.Browser):
    """A Zope ``testbrowser`` Browser that uses the Zope Publisher.

    The instance must set a custom user agent string.
    """

    def __init__(self, user_agent, url=None):

        mech_browser = PublisherMechanizeBrowser()
        mech_browser.addheaders = [("User-agent", user_agent),]

        # override the http handler class
        mech_browser.handler_classes["http"] = PublisherHTTPHandler
        browser.Browser.__init__(self, url=url, mech_browser=mech_browser)
```

For more information, see

- <https://mail.zope.org/pipermail/zope3-users/2008-May/007871.html>

Doctests

Doctests are way to do tests with interactive Python interpreter.

- <https://plone.org/documentation/tutorial/testing/doctests>

Doctests and pdb

Python debugger (pdb) works little differently when invoked from doctests.

Your locals stack frame is not what you might expect and refers to doctests internals:

```
(Pdb) locals()
{'__return__': None, 'self': <zope.testing.doctest._OutputRedirectingPdb instance at 0x5a7c8f0>}
```

Corrective action is to go one level up in the stack:

```
(Pdb) up
> /Users/moo/mmaspecial/src/Products.PloneGetPaid/Products/PloneGetPaid/notifications.py(22) __call__()
-> import pdb ; pdb.set_trace()
(Pdb) locals()
{'settings': <Products.PloneGetPaid.preferences.StoreSettings object at 0x5f631b0>,
 'store_url': 'http://nohost/plone', 'self': <Products.PloneGetPaid.notifications.MerchantOrderNotificationMessage object at 0x56c30d0>, 'order_contents': u'11 pz @84.00 total: US$924.00\n22 ph @59.00 total: US$1298.00\n12 pf @98.00 total: US$1176.00\n23 pX @95.00 total: US$2185.00\n3 pM @89.00 total: US$267.00\n22 po @60.00 total: US$1320.00\n23 pj @39.00 total: US$897.00\n15 po @34.00 total: US$510.00\n5 pS @76.00 total: US$380.00\n1 pm @70.00 total: US$70.00', 'template': u'To: $ {to_email}\nFrom: "${from_name}" <${from_email}>\nSubject: New Order\nNotification\n\nA New Order has been created\n\nTotal Cost: ${total_price}\n\nTo: continue processing the order follow this link:\n${store_url}/@@admin-manage-order/${order_id}/@@admin\n\nOrder Contents\n\n${order_contents}\n\nShipping Cost: $ {shipping_cost}\n\n', 'pdb': <module 'pdb' from '/Users/moo/code/python-macosx/parts/opt/lib/python2.4/pdb.pyc'>}
```

Interlude

Interlude is a Python package, which you can use to start an interactive Python shell from doctests, bypassing the limitations described above.

Just depend on ‘interlude’ and pass it via the globs dict to the doctest or import it from there:

```
>>> from interlude import interact
>>> interact(locals())
```

When the testrunner passes interact, you’ll get an interactive Python prompt.

For more information see: <https://pypi.python.org/pypi/interlude>

Get fields from browser

The most common operation when using a doctest is filling fields of a form:

```
>>> browser.getControl(name='form.widgets.text').value = 'Some text'
```

One common problem with this is that you can get an `LookupError`: `name ...`. If there is a typo, or the field does not exist, etc etc.

A quick way to see which fields exist on the current browser helps a lot while debugging test failures:

```
>>> [[c.name for c in f.controls] for f in browser.mech_browser.forms()]
```

Testing Cookbook

These are test snippets useful for common use cases.

General snippets

Test portal title:

```
def test_portal_title(self):
    self.assertEqual("Plone site", self.portal.getProperty('title'))
```

Test if view is protected:

```
def test_view_is_protected(self):
    from AccessControl import Unauthorized
    self.logout()
    with self.assertRaises(Unauthorized):
        self.portal.restrictedTraverse('@@view-name')
```

Test if object exists in folder:

```
def test_object_in_folder(self):
    self.assertNotIn('object_id', self.portal.objectIds())
```

JavaScript registered:

```
def test_js_available(self):
    jsreg = getattr(self.portal, 'portal_javascripts')
    script_ids = jsreg.getResourceIds()
    self.assertIn('my-js-file.js', script_ids)
```

CSS registered:

```
def test_css_available(self):
    cssreg = getattr(self.portal, 'portal_css')
    stylesheets_ids = cssreg.getResourceIds()
    self.assertIn('MyCSS.css', stylesheets_ids)
```

Test that a certain skin layer is present in portal_skins:

```
def test_skin_layer_installed(self):
    self.assertIn('my-skin-layer', self.skins.objectIds())
    self.assertIn('attachment_widgets', self.skins.objectIds())
```

Testing a clean uninstall

Description

How to test that your Plone add-on uninstalls cleanly

Introduction

Clean uninstall means that removing your add-on does not leave Plone site to broken state. Sometimes damage might not be noticed immediately, causing great frustration for the users.

Clean uninstall procedure is

- Use `Add on installer` to uninstall any add-ons. This **MUST** remove all add-on Python objects from the site ZODB database
- Remove eggs from the buildout, rerun buildout

If there are any Python objects, which classes come from the removed egg, around the site cannot be exported or imported anymore. Also, Plone upgrade might fail.

Clean uninstall test procedure

Manual procedure

- Create a Plone site from buildout, with your add-on egg present
- Install your add-on
- Play around with add-on to make sure it stores all its data (settings, local utilities, annotations, etc.)
- Uninstall add-on
- Export Plone site through the Management Interface as `zexp`
- Create another Plone site from vanilla buildout (no any add-ons installed)
- Import `.zexp`
- If `.zexp` does not contain any objects from your add-on egg, which is missing in vanilla Plone installation, your add-on installs cleanly

Example unit test

This code shows how to test that certain `:doc'annotations </components/annotations>'` are correctly cleaned.

Example:

```
"""
    Check that the site is clean after uninstall.
"""

__license__ = "GPL 2"
__copyright__ = "2009-2011 mFabrik Research Oy"

import unittest

from zope.component import getUtility, queryUtility, queryMultiAdapter

from Products.CMFCore.utils import getToolByName
from Products.CMFPlone.utils import get_installer

from gmobile.mobile.tests.base import BaseTestCase
from gmobile.mobile.behaviors import IMobileBehavior, mobile_behavior_factory,
↳ MobileBehaviorStorage
```

```
from zope.annotation.interfaces import IAnnotations

class TestUninstall(BaseTestCase):
    """ Test UA sniffing functions """

    def make_some_evil_site_content(self):
        """
        Add annotations etc. around the site
        """

        self.loginAsPortalOwner()
        self.portal.invokeFactory("Document", "doc")
        doc = self.portal.doc

        behavior = IMobileBehavior(doc)
        behavior.mobileFolderListing = False
        behavior.save()

        annotations = IAnnotations(doc)

    def uninstall(self, name="gomobile.mobile"):
        qi = get_installer(self.portal)

        try:
            qi.uninstall_product(name)
        except:
            pass
        qi.install_product(name)

    def test_annotations(self):
        """ Check that uninstaller cleans up annotations from the docs
        """
        self.make_some_evil_site_content()
        self.uninstall()

        annotations = IAnnotations(self.portal.doc)
        self.assertFalse("mobile" in annotations)

def test_suite():
    suite = unittest.TestSuite()
    suite.addTest(unittest.makeSuite(TestUninstall))
    return suite
```

plone.app.testing documentation

plone.testing documentation

plone.app.robotframework documentation

Developing for Plone Core follows similar patterns, but it requires you to sign the Plone Contributor license. The process is documented [here](#).

Writing proper code and documentation that others can expand upon is vital. As Plone community, we stick to the following style guides, and ask that all developers and documentation writers do the same.

Styleguides

These are styleguides for working on Plone and the Plone ecosystem.

They not only apply to code, but also documentation, naming conventions and other areas.

Python styleguide

Introduction

We've modeled the following rules and recommendations based on the following documents:

- [PEP8](#)
- [PEP257](#)
- [Rope project](#)
- [Google Style Guide](#)
- [Pylons Coding Style](#)
- [Tim Pope on Git commit messages](#)

Line length

All Python code in this package should be PEP8 valid. This includes adhering to the 80-char line length. If you absolutely need to break this rule, append `# noPEP8` to the offending line to skip it in syntax checks.

Note: Configuring your editor to display a line at 79th column helps a lot here and saves time.

Note: The line length rule also applies to non-python source files, such as `.zcm1` files, but is a bit more relaxed there. It explicitly **does not** apply to documentation `.rst` files. For `.rst` files including the package documentation but also `README.rst`, `CHANGES.rst` and doctests, use *semantic* linebreaks and add a line break after each sentence. See the [REST styleguide](#) for the reasoning behind it.

Breaking lines

Based on code we love to look at (Pyramid, Requests, etc.), we allow the following two styles for breaking long lines into blocks:

1. Break into next line with one additional indent block.

```
foo = do_something(  
    very_long_argument='foo', another_very_long_argument='bar')  
  
# For functions the ): needs to be placed on the following line  
def some_func(  
    very_long_argument='foo', another_very_long_argument='bar'):
```

```
very_long_argument='foo', another_very_long_argument='bar'
):
```

2. If this still doesn't fit the 80-char limit, break into multiple lines.

```
foo = dict(
    very_long_argument='foo',
    another_very_long_argument='bar',
)

a_long_list = [
    "a_fairly_long_string",
    "quite_a_long_string_indeed",
    "an_exceptionally_long_string_of_characters",
]
```

- Arguments on first line, directly after the opening parenthesis are forbidden when breaking lines.
- The last argument line needs to have a trailing comma (to be nice to the next developer coming in to add something as an argument and minimize VCS diffs in these cases).
- The closing parenthesis or bracket needs to have the same indentation level as the first line.
- Each line can only contain a single argument.
- The same style applies to dicts, lists, return calls, etc.

autopep8

Making old code pep8 compliant can be a lot of work. There is a tool that can automatically do some of this work for you: [autopep8](#). This fixes various issues, for example fixing indentation to be a multiple of four. Just install it with pip and call it like this:

```
pip install autopep8
autopep8 -i filename.py
autopep8 -i -r directory
```

It is best to first run autopep8 in the default non aggressive mode, which means it only does whitespace changes. To run this recursively on the current directory, changing files in place:

```
autopep8 -i -r .
```

Quickly check the changes and then commit them.

WARNING: be *very* careful when running this in a skins directory, if you run it there at all. It will make changes to the top of the file like this, which completely breaks the skin script:

```
-##parameters=policy_in=''
+# parameters=policy_in=''
```

With those safe changes out of the way, you can move on to a second, more aggressive round:

```
autopep8 -i --aggressive -r .
```

Check these changes more thoroughly. At the very least check if Plone can still start in the foreground and that there are no failures or errors in the tests.

Not all changes are always safe. You can ignore some checks:

```
autopep8 -i --ignore W690,E711,E721 --aggressive -r .
```

This skips the following changes:

- W690: Fix various deprecated code (via lib2to3). (Can be bad for Python 2.4.)
- E721: Use *isinstance()* instead of comparing types directly. (There are uses of this in for example GenericSetup and plone.api that must not be fixed.)
- E711: Fix comparison with None. (This can break SQLAlchemy code.)

You can check what would be changed by one specific code:

```
autopep8 --diff --select E309 -r .
```

Indentation

For Python files, we stick with the [PEP 8 recommendation](#): Use 4 spaces per indentation level.

For ZCML and XML (GenericSetup) files, we recommend the [Zope Toolkit's coding style on ZCML](#):

```
Indentation of 2 characters to show nesting, 4 characters to list attributes on
↳ separate lines.
This distinction makes it easier to see the difference between attributes and nested
↳ elements.
```

Quoting

For strings and such prefer using single quotes over double quotes. The reason is that sometimes you do need to write a bit of HTML in your python code, and HTML feels more natural with double quotes so you wrap HTML string into single quotes. And if you are using single quotes for this reason, then be consistent and use them everywhere.

There are two exceptions to this rule:

- docstrings should always use double quotes (as per PEP-257).
- if you want to use single quotes in your string, double quotes might make more sense so you don't have to escape those single quotes.

```
# GOOD
print 'short'
print 'A longer string, but still using single quotes.'

# BAD
print "short"
print "A long string."

# EXCEPTIONS
print "I want to use a 'single quote' in my string."
"""This is a docstring."""
```

Docstrings style

Read and follow <http://www.python.org/dev/peps/pep-0257/>. There is one exception though: We reject BDFL's recommendation about inserting a blank line between the last paragraph in a multi-line docstring and its closing quotes as it's Emacs specific and two Emacs users here on the Beer & Wine Sprint both support our way.

The content of the docstring must be written in the active first-person form, e.g. “Calculate X from Y” or “Determine the exact foo of bar”.

```
def foo():
    """Single line docstring."""

def bar():
    """Multi-line docstring.

    With the additional lines indented with the beginning quote and a
    newline preceding the ending quote.
    """
```

If you wanna be extra nice, you are encouraged to document your method’s parameters and their return values in a reST field list syntax.

```
:param foo: blah blah
:type foo: string
:param bar: blah blah
:type bar: int
:returns: something
```

Check out the [plone.api](#) source for more usage examples. Also, see the following for examples on how to write good *Sphinx* docstrings: <http://stackoverflow.com/questions/4547849/good-examples-of-python-docstrings-for-sphinx>.

Unit tests style

Read <http://www.voidspace.org.uk/python/articles/unittest2.shtml> to learn what is new in `unittest2` and use it.

This is not true for in-line documentation tests. Those still use old `unittest` test-cases, so you cannot use `assertIn` and similar.

String formatting

As per <http://docs.python.org/2/library/stdtypes.html#str.format>, we should prefer the new style string formatting (`.format()`) over the old one (`% ()`).

Also use numbering, like so:

```
# GOOD
print "{0} is not {1}".format(1, 2)
```

and *not* like this:

```
# BAD
print "{} is not {}".format(1, 2)
print "%s is not %s" % (1, 2)
```

because Python 2.6 supports only explicitly numbered placeholders.

About imports

1. Don’t use `*` to import *everything* from a module, because if you do, `pyflakes` cannot detect undefined names (W404).

2. Don't use commas to import multiple things on a single line. Some developers use IDEs (like [Eclipse](#)) or tools (such as [mr.igor](#)) that expect one import per line. Let's be nice to them.
3. Don't use relative paths, again to be nice to people using certain IDEs and tools. Also *Google Python Style Guide* recommends against it.

```
# GOOD
from plone.app.testing import something
from zope.component import getMultiAdapter
from zope.component import getSiteManager
```

instead of

```
# BAD
from plone.app.testing import *
from zope.component import getMultiAdapter, getSiteManager
```

4. Don't catch `ImportError` to detect whether a package is available or not, as it might hide circular import errors. Instead, use `pkg_resources.get_distribution` and catch `DistributionNotFound`. More background at http://do3.cc/blog/2010/08/20/do-not-catch-import-errors,-use-pkg_resources/.

```
# GOOD
import pkg_resources

try:
    pkg_resources.get_distribution('plone.dexterity')
except pkg_resources.DistributionNotFound:
    HAS_DEXTERITY = False
else:
    HAS_DEXTERITY = True
```

instead of

```
# BAD
try:
    import plone.dexterity
    HAVE_DEXTERITY = True
except ImportError:
    HAVE_DEXTERITY = False
```

Grouping and sorting

Since Plone has such a huge code base, we don't want to lose developer time figuring out into which group some import goes (standard lib?, external package?, etc.). We sort everything alphabetically case insensitive and insert one blank line between `from foo import bar` and `import baz` blocks. Conditional imports come last. Don't use multi-line imports but import each identifier from a module in a separate line. Again, we *do not* distinguish between what is standard lib, external package or internal package in order to save time and avoid the hassle of explaining which is which.

```
# GOOD
from __future__ import division
from Acquisition import aq_inner
from datetime import datetime
from datetime import timedelta
from plone.api import portal
from plone.api.exc import MissingParameterError
```

```
from Products.CMFCore.interfaces import ISiteRoot
from Products.CMFCore.WorkflowCore import WorkflowException

import pkg_resources
import random

try:
    pkg_resources.get_distribution('plone.dexterity')
except pkg_resources.DistributionNotFound:
    HAS_DEXTERITY = False
else:
    HAS_DEXTERITY = True
```

`isort`, a python tool to sort imports can be configured to sort exactly as described above.

Add the following:

```
[settings]
force_alphabetical_sort = True
force_single_line = True
lines_after_imports = 2
line_length = 200
not_skip = __init__.py
```

To either `.isort.cfg` or changing the header from `[settings]` to `[isort]` and putting it on `setup.cfg`.

You can also use `plone.recipe.codeanalysis` with the `flake8-isort` plugin enabled to check for it.

Declaring dependencies

All direct dependencies should be declared in `install_requires` or `extras_require` sections in `setup.py`. Dependencies, which are not needed for a production environment (like “develop” or “test” dependencies) or are optional (like “Archetypes” or “Dexterity” flavors of the same package) should go in `extras_require`. Remember to document how to enable specific features (and think of using `zcml:condition` statements, if you have such optional features).

Generally all direct dependencies (packages directly imported or used in ZCML) should be declared, even if they would already be pulled in by other dependencies. This explicitness reduces possible runtime errors and gives a good overview on the complexity of a package.

For example, if you depend on `Products.CMFPlone` and use `getToolByName` from `Products.CMFCore`, you should also declare the `CMFCore` dependency explicitly, even though it’s pulled in by Plone itself. If you use namespace packages from the Zope distribution like `Products.Five` you should explicitly declare Zope as dependency.

Inside each group of dependencies, lines should be sorted alphabetically.

Versioning scheme

For software versions, use a sequence-based versioning scheme, which is compatible with `setuptools`:

```
MAJOR.MINOR[.MICRO][.STATUS]
```

The way, `setuptools` interprets versions is intuitive:

```
1.0 < 1.1.dev < 1.1.a1 < 1.1.a2 < 1.1.b < 1.1.rc1 < 1.1 < 1.1.1
```


You can test it with `setuptools`:

```
>>> from pkg_resources import parse_version
>>> parse_version('1.0') < parse_version('1.1.dev')
... < parse_version('1.1.a1') < parse_version('1.1.a2')
... < parse_version('1.1.b') < parse_version('1.1.rc1')
... < parse_version('1.1') < parse_version('1.1.1')
True
```

`dev` and `dev0` are treated as the same:

```
>>> parse_version('1.1.dev') == parse_version('1.1.dev0')
True
```

`Setuptools` recommends to separate parts with a dot. The website about [semantic versioning](#) is also worth a read.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't “require” PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare `except:` `pass` will likely be reverted instantly.
- Avoid `try: on-error,` since this swallows exceptions.
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the [ZODB level](#)!
- Never put any HTML in Python code and return it as a string. There are exceptions, though.
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use [Views](#) instead!)
- Remember to add [i18n](#) tags in ZPTs and Python code.

JavaScript styleguide

Note: Plone doesn't yet use any of the new [ES2015](#) features.

Many of the style guide recommendations here come from Douglas Crockford's seminal book [JavaScript, the good parts](#).

Indentation

Indentation is an important aid for readability and comprehension. When editing a file, please keep to the convention already established.

In [Patternslib](#) we indent 4 spaces as suggested by Douglas Crockford in *JavaScript, the good parts*.

The [Mockup](#) patterns on the other hand we indent 2 spaces.

Naming of variables, classes and functions

Underscores or camelCase?

We use camelCase for function names and underscores_names for variables names.

For example:

```
function thisIsAFunction () {
    var this_is_a_variable;
    ...
}
```

jQuery objects are prefixed with \$

We prefix jQuery objects with the \$ sign, to distinguish them from normal DOM elements.

For example:

```
var divs = document.getElementsByTagName('div'); // List of DOM elements
var $divs = $('div'); // jQuery object
```

Spaces around operators

In general, spaces are put around operators, such as the equals = or plus + signs.

For example:

```
if (sublocale != locale) {
    // do something
}
```

An exception is when they appear inside for-loop expressions, for example:

```
for (i=0; i<msgs_length; i++) {
    // do something
}
```

Generally though, rather err on the side of adding spaces, since they make the code much more readable.

Constants are written in ALL_CAPS

Identifiers that denote constant values should be written in all capital letters, with underscores between words.

For example:

```
var SECONDS_IN_HOUR = 3600; // constant
var seconds_since_click = 0; // variable
```

Function declaration and invocation

In his book, *JavaScript, the good parts*, Douglas Crockford suggests that function names and the brackets that come afterwards should be separated with a space, to indicate that it's a declaration and not a function call or instantiation.

```
function update (model) { // function declaration
    model.foo = 'bar';
}

update(model); // function call
```

This practice however doesn't appear to be very common and is also not used consistently throughout the codebase. It might however be useful sometimes, to reduce confusion.

Checking for equality

JavaScript has a strict `===` and less strict `==` equality operator. The stricter operator also does type checking. To avoid subtle bugs when doing comparisons, always use the strict equality check.

Curly brackets

Curly brackets must appear on the same lines as the `if` and `else` keywords. The closing curly bracket appears on its own line.

For example:

```
if (locale) {
    return locales[locale];
} else {
    sublocale = locale.split("-")[0];
    if (sublocale != locale && locales[sublocale]) {
        return locales[sublocale];
    }
}
```

Always enclose blocks in curly brackets

When writing an a block such as an `if` or `while` statement, always use curly brackets around that block of code. Even when not strictly required by the compiler (for example if its only one line inside the `if` statement).

For example, like this:

```
if (condition === true) {
    this.updateRoomsList();
}
somethingElse();
```

and **NOT** like this:

```
if (condition === true)
    this.updateRoomsList();
somethingElse();
```

This is to aid in readability and to avoid subtle bugs where certain lines are wrongly assumed to be executed within a block.

Binding the “this” variable versus assigning to “self”

One of the deficiencies in JavaScript is that callback functions are not bound to the correct or expected context (as referenced with the `this` variable). In [ES2015](#), this problem is solved by using so-called arrow functions for callbacks.

However, while we’re still writing ES5 code, we can use the `.bind` method to bind the correct `this` context to the callback method.

For example:

```
this.$el = $("#some-element");
setTimeout(function () {
    // Without using .bind, "this" will refer to the window object.
    this.$el.hide();
}.bind(this), 1000);
```

What about assigning the outer “this” to “self”?

A different way of solving the above problem is to assign the outer `this` variable to `self` and then using `self` in the callback.

For example:

```
var self = this;
self.$el = $("#some-element");
setTimeout(function () {
    self.$el.hide();
}, 1000);
```

This practice is commonly used in the Mockup patterns.

It is however discouraged in Patternslib because it results in much longer functions due to the fact that callback functions can’t be moved out of the containing function where `self` is defined.

Additionally, `self` is by default an alias for `window`. If you forget to use `var self`, there’s the potential for bugs that can be difficult to track down.

Douglas Crockford and others suggest that the variable `that` be used instead, which is also the convention we follow in Patternslib.

For example:

```
var that = this;
that.$el = $("#some-element");
setTimeout(function () {
    that.$el.hide();
}, 1000);
```

Use named functions

JavaScript has both named functions and unnamed functions.

```
// This is a function named "foo"
function foo() { }

// This is an unnamed function
var foo = function() { };
```

Unnamed functions are convenient, but result in unreadable call stacks and profiles. This makes debugging and profiling code unnecessarily hard. To fix this always use named functions for non-trivial functions.

```
$el.on("click", function buttonClick(event) {
    ...
});
```

An exception to this rule are trivial functions that do not call any other functions, such as functions passed to `Array.filter` or `Array.forEach`.

Pattern methods must always be named, and the name should be prefixed with the pattern name to make them easy to recognize.

```
var mypattern = {
    name: "mypattern",

    init: function mypatternInit($el) { },
    _onClick: function mypatternOnClick(e) { }
};
```

Custom events

A pattern can send custom events for either internal purposes, or as a hook for third party JavaScript. Since IE8 is still supported `CustomEvent` can not be used. Instead you must send custom events using `jQuery's trigger function`. Event names must follow the `pat-<pattern name>-<event name>` pattern.

```
$(el).trigger("pat-tooltip-open");
```

The element must be dispatched from the element that caused something to happen, *not* from the elements that are changed as a result of an action.

All extra data must be passed via a single object. In a future Patterns release this will be moved to the `detail` property of a `CustomEvent` instance.

```
$(el).trigger("pat-toggle-toggled", {value: new_value});
```

Event listeners can access the provided data as an extra parameter passed to the event handler.

```
function onToggled(event, detail) {
}
$(".myclass").on("pat-toggle-toggled", onToggled);
```

Event listeners

All event listeners registered using `jQuery.fn.on` must be namespaced with `pat-<pattern name>`.

```
function mypattern_init($el) {
    $el.on("click.pat-mypattern", mypattern._onClick);
}
```

Storing arbitrary data

When using `jQuery.fn.data` the storage key must either be `pat-<pattern name>` if a single value is stored, or `pat-<pattern name>-<name>` if multiple values are stored. This prevents conflicts with other code.

```
// Store parsed options
$(el).data("pat-mypattern", options);
```

Styleguide for documentation

General style guides on documentation

For general information on how to write documentation, see the [documentation styleguide](#). Information on our Restructured Text style guide can be found in the: [REST styleguide](#).

Restructured Text versus Plain Text

Use the Restructured Text (`.rst` file extension) format instead of plain text files (`.txt` file extension) for all documentation, including doctest files. This way you get nice syntax highlighting and formatting in recent text editors, on GitHub and with Sphinx.

Tracking changes

Feature-level changes to code are tracked inside `CHANGES.rst`. The title of the `CHANGES.rst` file should be Changelog. Example:

```
Changelog
=====

1.0.0-dev (Unreleased)
-----

- Added feature Z.
  [github_userid1]

- Removed Y.
  [github_userid2]

1.0.0-alpha.1 (2012-12-12)
-----

- Fixed Bug X.
  [github_userid1]
```

Add an entry every time you add/remove a feature, fix a bug, etc. on top of the current development changes block.

Git commit message style guide

Tim Pope's post on Git commit message style is widely considered the gold standard:

Capitalized, short (50 chars or less) summary

More detailed explanatory text, if necessary. Wrap it to about 72 characters or so. In some contexts, the first line is treated as the subject of an email and the rest of the text as the body. The blank line separating the summary from the body is critical (unless you omit the body entirely); tools like rebase can get confused if you run the two together.

Write your commit message in the imperative: "Fix bug" and not "Fixed bug" or "Fixes bug." This convention matches up with commit messages generated by commands like git merge and git revert.

Further paragraphs come after blank lines.

- Bullet points are okay, too
- Typically a hyphen or asterisk is used for the bullet, preceded by a single space, with blank lines in between, but conventions vary here
- Use a hanging indent

GitHub flavored markdown is also useful in commit messages.

Naming Conventions

Above all else, be consistent with any code you are modifying!

Historically the code is all camel case, but new code should be written following the PEP8 convention.

Class names should be written in CamelCase and function and method names all lowercase with an underscore separating words (e.g. my_method).

File Conventions

In Zope 2 file names used to be MixedCase. In Python and Plone we prefer all-lowercase filenames.

This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplate
```

Filenames should be short and descriptive.

Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is better to read, less redundant and generally more aesthetically pleasing.

Plone Deprecation Guide

Introduction

This document describes rationales, configuration and best practices of deprecations in Plone, Zope and Python. It is meant as a styleguide on how to apply deprecations in Plone core packages. It also has a value as a general overview on how to deprecate in Python.

Why Deprecation

At some point we

- need to get rid of old code,
- want to unify api style (consistent api),
- fix typos in namings,
- move code around (inside package or to another package).

While refactoring code, moving modules, functions, classes and methods is often needed. To not break third party code imports from the old place or usage of old functions/ methods must work for while. Deprecated methods are usually removed with the next major release of Plone. Following the [semantic versioning guideline](#) is recommended.

Help Programmers, No annoyance

Deprecation has to support the consumers of the code - the programmers using it. From their point of view, Plone core code is an API to them. Any change is annoying to them anyway, but they feel better if deprecation warnings are telling them what to do.

Deprecations must always log at level *warning* and have to answers the question:

“Why is the code gone from the old place? What to do instead?”

A short message is enough., i.e.:

- “Replaced by new API xyz, found at abc.cde”.,
- “Moved to xyz, because of abc.”,
- “Name had a typo, new name is “xyz”.

All logging has to be done once, i.e. on first usage or first import. It must not flood the logs.

Use Cases

Renaming We may want to rename classes, methods, functions or global or class variables in order to get a more consistent api or because of a typo, etc. We never just rename, we always provide a deprecated version logging a verbose deprecation warning with information where to import from in future.

Moving a module, class, function, etc to another place For some reason, i.e. merging packages, consistent api or resolving circular import problems, we need to move code around. When imported from the old place it logs a verbose deprecation warning with information where to import from in future.

Deprecation of a whole package A whole package (folder with `__init__.py`)

- all imports still working, logging deprecation warnings on first import

- ZCML still exists, but is empty (or includes the zcml from the new place if there's no auto import (i.e. for meta.zcml).

Deprecation of a whole python egg We will provide a last major release with no ‘real’ code, only backward compatible (bbb) imports of public API are provided. This will be done the way described above for a whole package. The README clearly states why it was moved and where to find the code now.

Deprecation of a GenericSetup profile They may get renamed for consistency or are superfluous after an update. Code does not need to break to support this.

Enable Deprecation Warnings

Zope

Zope does configure logging and warnings, so the steps below (under section Python) are not needed.

Using `plone.recipe.zope2instance` add the option `deprecation-warnings = on` to the `buildouts [instance]` section.

```
[buildout]
parts = instance

[instance]
recipe = plone.recipe.zope2instance
...
deprecation-warnings = on
...
```

This just sets a configuration option in `zope.conf`.

Without the recipe this can be set manually as well: In `zope.conf` custom filters for warnings can be defined.

```
...
<warnfilter>
    action always
    category exceptions.DeprecationWarning
</warnfilter>
...
```

Python

Enable Warnings Warnings are written to `stderr` by default, but `DeprecationWarning` output is suppressed by default.

Output can be enabled by starting the Python interpreter with the `-W [all|module|once]` option.

It is possible to enable output in code too:

```
import warnings
warnings.simplefilter("module")
```

Configure Logging Once output is enabled it is possible to [redirect warnings to the logger](#):

```
import logging
logging.captureWarnings(True)
```

Running tests

In Plone tests deprecation warnings are not shown by default. The `zope.conf` setting is not taken into account.

In order to enable deprecation warnings, the Python way with the `-W` command option must to be used.

Given youre using a modern buildout with virtualenv as recommended, the call looks like so:

```
./bin/python -W module ./bin/test
```

Deprecation Best Practice

Vanilla Deprecation Messages

Python offers a built-in `DeprecationWarning` which can be issued using standard libraries `warnings` module.

For details read the [official documentation about warnings](#).

In short it works like so

```
import warnings
warnings.warn('deprecated', DeprecationWarning)
```

Moving Whole Modules

Given a package `old.pkg` with a module `foo.py` need to be moved to a package `new.pkg` as `bar.py`.

`zope.deprecation` [Moving modules](#) offers a helper.

1. Move the `foo.py` as `bar.py` to the `new.pkg`.
2. At the old place create a new `foo.py` and add to it

```
from zope.deprecation import moved
moved('new.pkg.bar', 'Version 2.0')
```

Now you can still import the namespace from `bar` at the old place, but get a deprecation warning:

```
DeprecationWarning: old.pkg.foo has moved to new.pkg.bar. Import of old.pkg.foo will become unsupported in Version 2.0
```

Moving Whole Packages

This is the same as moving a module, just create for each module a file.

Deprecating methods and properties

You can use the `@deprecated` decorator from `zope.deprecation` [Deprecating methods and properties](#) to deprecate methods in a module:

```
from zope.deprecation import deprecated

@deprecated('Old method is no longer supported, use new_method instead.')
```

```
def old_method():
    return 'some value'
```

The deprecated wrapper method is for deprecating properties:

```
from zope.deprecation import deprecated

foo = None
foo = deprecated(foo, 'foo is no more, use bar instead')
```

Moving functions and classes

Given we have a Python file at `old/foo/bar.py` and want to move some classes or functions to `new/baz/baaz.py`.

Here `zope.deferredimport` offers a deprecation helper. It also avoids circular imports on initialization time.

```
import zope.deferredimport
zope.deferredimport.initialize()

zope.deferredimport.deprecated(
    "Import from new.baz.baaz instead",
    SomeOldClass='new.baz:baaz.SomeMovedClass',
    some_old_function='new.baz:baaz.some_moved_function',
)

def some_function_which_is_not_touched_at_all():
    pass
```

Deprecating a GenericSetup profile

Starting with GenericSetup 1.8.2 (part of Plone > 5.0.2) the `post_handler` attribute in ZCML can be used to call a function after the profile was applied. We use this feature to issue a warning.

First we register the same profile twice. Under the new name and under the old name:

```
<genericsetup:registerProfile
    name="default"
    title="My Fance Package"
    directory="profiles/default"
    description="..."
    provides="Products.GenericSetup.interfaces.EXTENSION"
/>

<genericsetup:registerProfile
    name="some_confusing_name"
    title="My Fance Package (deprecated)"
    directory="profiles/some_confusing_name"
    description="... (use profile default instaed)"
    provides="Products.GenericSetup.interfaces.EXTENSION"
    post_handler=".setuphandlers.deprecate_profile_some_confusing_name"
/>
```

And in `setuphandlers.py` add a function:

```
import warnings

def deprecate_profile_some_confusing_name(tool):
    warnings.warn(
        'The profile with id "some_confusing_name" was renamed to "default".',
        DeprecationWarning
    )
```

How to set up your editor

`EditorConfig` provides a way to share the same configuration for all major source code editors.

You only need to install the plugin for your editor of choice, and add the following configuration on `~/.editorconfig`.

```
[*]
indent_style = space
end_of_line = lf
insert_final_newline = true
trim_trailing_whitespace = true
charset = utf-8

[*.py,*.cfg]
indent_size = 4

[*.html,*.dtml,*.pt,*.zpt,*.xml,*.zcml,*.js]
indent_size = 2

[Makefile]
indent_style = tab
```

dotfiles

Some Plone developers use dotfiles similar to these: [plone.dotfiles](#). This might inspire you with your own dotfiles/configuration settings.

Some of the information here is taken from the `plone.api` contributing guide

Importing content from other systems often requires the help of tools to get content out from various sources and into Plone. A number of these tools exist.

Importing content from other sources

Description

There are various tools to help you import content from other systems into Plone

Introduction

Rarely does a new website start all from scratch. Most of the time, you will have to import content from other systems. These may include:

- other CMS systems, sometimes based on PHP/MySQL
- legacy sites in plain HTML
- resources that exist on a filesystem, such as files and images
- other Plone sites, including older and unmaintained versions

While Plone even comes with an FTP service, that can serve as a last-ditch effort to get some pictures in, there are far more sophisticated tools available.

Transmogrify

By far the most flexible tool available is something called **collective.transmogrifier**.

Note: A transmogrifier is fictional device used for transforming one object into another object. The term was coined by Bill Waterson of Calvin and Hobbes fame.

In principle, what it does is to allow you to lay a ‘pipeline’, whereby an object (a piece of content) is transported. At each part of the pipeline, you can perform various operations on it: extract, change, add metadata, etcetera. These operations are in the form of so-called ‘blueprints’.

In short: an object is gathered from a source you define. Then, it goes to one or more segments of the pipeline to let the various blueprints work on it, and in the end you use a ‘constructor’ to turn it into a Plone content object.

That’s the basics, but by combining all your options you have an incredibly flexible and powerful tool at hand.

collective.transmogrifier

See the extensive documentation:

Transmogrify helpers

Various add-ons exist to make working with transmogrify easier:

- [mr.migrator](#) is a way to lay pipelines
- [funnelweb](#) helps to parse static sites, and crawl external sites
- [parse2plone](#) is meant to get HTML content from the file system into Plone

And a wide array of extra ‘blueprints’ exist, like

- [quintagroup.transmogrifier](#)
- [transmogrify.sqlalchemy](#) to get content out of just about any SQL database you can think of
- [collective.jsonmigrator](#) is good at migrating data via JSON format from very old Plone versions, going back all the way to 2.x

Note this is only a selection, do a search on pypi to find more. NB searching both on [transmogrify](#) and [transmogrifier](#) gives more results!

Tutorials

“Mastering Plone”-training

Mastering Plone is intended as a week-long training for people who are new to Plone or want to learn about the current best-practices of Plone-development.

It is in active use by various trainers in the Plone world, and is being developed as a ‘collaborative syllabus’.

And while attending one of the trainings with real trainers is the best thing to do, you can learn a great deal from following the documentation for these trainings.

“Mastering Mockup”-training

This training was created to teach about Mockup, the new Frontend library for Plone 5 .

Selected Plone core package documentation

plone.api

plone.api is the recommended way of accessing Plone’s functionality in your own code.

plone.app.multilingual

The default solution to create multilingual content.

plone.app.contenttypes

The default dexterity-based content types, since Plone 5.

plone.app.contentlisting

plone.app.event

The calendar framework for Plone, default since Plone 5.

Glossary

This is a glossary for some definitions used in this documentation and still under construction.

.po The file format used by the *gettext* translation system. <http://www.gnu.org/software/hello/manual/gettext/PO-Files.html>

Acquisition Simply put, any Zope object can acquire any object or property from any of its parents. That is, if you have a folder called *A*, containing two resources (a document called *homepage* and another folder called *B*), then an URL pointing at *http://.../A/B/homepage* would work even though *B* is empty. This is because Zope starts to look for *homepage* in *B*, doesn't find it, and goes back up to *A*, where it's found. The reality, inevitably, is more complex than this. For the whole story, see the [Acquisition chapter in the Zope Book](#).

AGX AGX is short for *ArchGenXML*.

Archetypes Archetypes is a framework designed to facilitate the building of applications for Plone and *CMF*. Its main purpose is to provide a common method for building content objects, based on schema definitions. Fields can be grouped for editing, making it very simple to create wizard-like forms. Archetypes is able to do all the heavy lifting needed to bootstrap a content type, allowing the developer to focus on other things such as business rules, planning, scaling and designing. It provides features such as auto-generation of editing and presentation views. Archetypes code can be generated from *UML* using *ArchGenXML*.

ArchGenXML ArchGenXML is a code-generator for CMF/Plone applications (a *Product*) based on the *Archetypes* framework. It parses UML models in XMI-Format (*.xmi*, *.zargo*, *.zuml*), created with applications such as ArgoUML, Poseidon or ObjectDomain. A brief tutorial for ArchGenXML is present on the plone.org site.

ATCT ATContentTypes - the Plone content types written with Archetypes which replaces the default CMF content types in Plone 2.1 onwards.

BBB When adding (or leaving) a piece of code for backward compatibility, we use a BBB comment marker with a date.

browserview Plone uses a view pattern to output dynamically generated HTML pages. Views are the basic elements of modern Python web frameworks. A view runs code to setup Python variables for a rendering template.

Output is not limited to HTML pages and snippets, but may contain JSON, file download payloads, or other data formats. See [views](#)

Buildout Buildout is a Python-based build system for creating, assembling and deploying applications from multiple parts, some of which may be non-Python-based. It lets you create a buildout configuration and reproduce the same software later. See buildout.org

Catalog The catalog is an internal index of the content inside Plone so that it can be searched. The catalog object is accessible through the [ZMI](#) as the `portal_catalog` object.

CMF The *Content Management Framework* is a framework for building content-oriented applications within Zope. It as formed the basis of Plone content from the start.

Collective The *Collective* is a community code repository for Plone Products and other add-ons, and is a useful place to find the very latest code for hundreds of add-ons to Plone. Developers of new Plone Products are encouraged to share their code via the Collective so that others can find it, use it, and contribute fixes and improvements.

control panel The Control Panel is the place where many parameters of a Plone site can be set. Here add-ons can be enabled, users and groups created, the workflow and permissions can be set and settings for language, caching and many other can be found. If you have “Site Admin” permissions, you can find it under “Site -> Site Setup” in your personal tools.

CSS Cascading Style Sheets is a way to separate content from presentation. Plone uses this extensively, and it is a web standard [documented at the W3C web site](#). If you want to learn CSS, we recommend the [W3Schools CSS Resources](#) and the [SitePoint CSS Reference](#).

Dexterity Dexterity is an alternative to [Archetypes](#), Plone’s venerable content type framework. Being more recent, Dexterity has been able to learn from some of the mistakes that were made Archetypes, and - more importantly - leverage some of the technologies that did not exist when Archetypes was first conceived. Dexterity is built from the ground up to support through-the-web type creation. Dexterity also allows types to be developed jointly through-the-web and on the filesystem. For example, a schema can be written in Python and then extended through the web.

Diazo The standard way to theme Plone sites from Plone 5 onwards. It consists in essence of a static ‘theme’ mockup of your website, with HTML, CSS and JavaScript files, and a set of rules that will ‘switch in’ the dynamic content from Plone into the theme.

Document A document is a page of content, usually a self-contained piece of text. Documents can be written in several different formats, plain text, HTML or (re)Structured Text. The default home page for a Plone site is one example of a document.

DTML Document Template Markup Language. DTML is a server-side templating language used to produce dynamic pieces of content, but is now superseded by [ZPT](#) for HTML and XML content. It is still used sparingly for non-XML content like SQL and mail/CSS.

Dublin Core Dublin Core is a standard set of metadata which enables the description of resources for the purposes of discovery. See https://en.wikipedia.org/wiki/Dublin_Core

easy_install A command-line tool for automatic discovery and installation of packages into a Python environment. The `easy_install` script is part of the `setuptools` package, which uses the [Python Package Index](#) as its source for packages.

Egg See [Python egg](#).

Expiration Date The last day an item should show up in searches, news listings etc. Please note that this doesn’t actually remove or disable the item, it merely makes it not show up in searches.

This is part of the Dublin Core metadata that is present on all Plone objects.

GenericSetup An XML-based configuration system for Zope and Plone applications.

gettext UNIX standard software translation tool. See <http://www.gnu.org/software/gettext/>

i18n i18n is shorthand for “internationalization” (the letter I, 18 letters, the letter N) - and refers to the process of preparing a program so that it can be used in multiple languages without further altering the source. Plone is fully internationalized.

i18ndude Support tool to create and update message catalogs from instrumented source code.

JSON JavaScript Object Notation. JSON is a lightweight text-based open standard designed for human-readable data interchange. In short, it’s a string that looks like a JavaScript array, but is constrained to 6 simple data types. It can be parsed by many languages.

KSS *Kinetic Style Sheets* is a client-side framework for implementing rich user interfaces with AJAX functionality. It allows attaching actions to elements using a CSS-like rule syntax. KSS was added to Plone in Plone 3 and removed in Plone 4.3, because JQuery made it obsolete.

Kupu Kupu was the user-friendly graphical HTML editor component that used to be bundled with Plone, starting with version 2.1. It has since been replaced by *TinyMCE*.

l10n Localization is the actual preparing of data for a particular language. For example Plone is i18n aware and has localization for several languages. The term l10n is formed by the first and last letter of the word and the number of letters in between.

Layer A layer is a set of templates and scripts that get presented to the user. By combining these layers, you create what is referred to as a *skin*. The order of layers is important, the topmost layers will be examined first when rendering a page. Each layer is an entry in `portal_skins` -> ‘Contents’, and is usually a Filesystem Directory View or a Folder.

LDAP Lightweight Directory Access Protocol. An internet protocol which provides a specification for user-directory access by wire, attribute syntax, representation of distinguished names, search filters, an URL format, a schema for user-centric information, authentication methods, and transport layer security. Example: an email client might connect to an LDAP server in order to look up an email address for a person by a person’s name.

Manager The *Manager* Security role is a standard role in Zope. A user with the Manager role has ALL permissions except the Take Ownership permission. Also commonly known as Administrator or root in other systems.

METAL Macro Expansion Template Attribute Language. See *ZPT*.

Monkey patch A monkey patch is a way to modify the behavior of Zope or a Product without altering the original code. Useful for fixes that have to live alongside the original code for a while, like security hotfixes, behavioral changes, etc.

The term “monkey patch” seems to have originated as follows: First it was “guerrilla patch”, referring to code that sneakily changes other code at runtime without any rules. In Zope 2, sometimes these patches conflict. This term went around Zope Corporation for a while. People heard it as “gorilla patch”, though, since the two words sound very much alike, and the word gorilla is heard more often. So, when someone created a guerrilla patch very carefully and tried to avoid any battles, they tried to make it sound less forceful by calling it a monkey patch. The term stuck.

Namespace package A feature of setuptools which makes it possible to distribute multiple, separate packages sharing a single top-level namespace. For example, the packages `plone.theme` and `plone.portlets` both share the top-level `plone` namespace, but they are distributed as separate eggs. When installed, each egg’s source code has its own directory (or possibly a compressed archive of that directory). Namespace packages eliminate the need to distribute one giant plone package, with a top-level plone directory containing all possible children.

OpenID A distributed identity system. Using a single URI provider an individual is able to login to any web site that accepts OpenID using the URI and a password. Plone implements OpenID as a *PAS* plug-in.

PAS The Pluggable Authentication Service (PAS) is a framework for handling authentication in Zope 2. PAS is a Zope `acl_users` folder object that uses “plugins” that can implement various authentication interfaces (for example *LDAP* and *OpenID*) that plug into the PAS framework . Zope 3 also uses a design inspired by PAS. PAS was integrated into Plone at the 2005 San Jose Sprint.

PLIP *PLone Improvement Proposal* (just like Python’s PEPs: Python Enhancement Proposals). These are documents written to structure and organise proposals for the improvement of Plone.

Motivation, deliverables, risks and a list of people willing to do the work must be included. This document is submitted to the [Framework Team](#), who reviews the proposal and decides if it’s suitable to be included in the next Plone release or not.

See more info about how to write a [PLIP](#).

Plonista A Plonista is a member of the Plone community. It can be somebody who loves Plone, or uses Plone, or someone who spreads Plone and Plone knowledge. It can also be someone who is a Plone developer, or it can be all of the above.

Product A Plone-specific module that extends Plone functionality and can be managed via the Plone Control Panel. Plone Products often integrate non-Plone-specific modules for use within the Plone context.

Python egg A widely used Python packaging format which consists of a zip or `.tar.gz` archive with some metadata information. It was introduced by [setuptools](#)

A way to package and distribute Python packages. Each egg contains a `setup.py` file with metadata (such as the author’s name and email address and licensing information), as well as information about dependencies. `setuptools`, the Python library that powers the egg mechanism, is able to automatically find and download dependencies for eggs that you install. It is even possible for two different eggs to concurrently use different versions of the same dependency. Eggs also support a feature called *entry points*, a kind of generic plug-in mechanism.

Python package A general term describing a redistributable Python module. At the most basic level, a package is a directory with an `__init__.py` file, which can be blank.

Python Package Index The Python community’s index of thousands of downloadable Python packages. It is available as a website to browse, with the ability to search for a particular package. More importantly, `setuptools`-based packaging tools (most notably, `buildout` and `easy_install`) can query this index to download and install eggs automatically. Also known as the Cheese Shop or PyPI.

Python path The order and location of folders in which the Python interpreter will look for modules. It’s available in python via `sys.path`. When Zope is running, this typically includes the global Python modules making up the standard library, the interpreter’s site-packages directory, where third party “global” modules and eggs are installed, the Zope software home, and the `lib/python` directory inside the instance home. It is possible for python scripts to include additional paths in the Python path during runtime. This ability is used by `zc.buildout`.

RAD Rapid Application Development - A term applied to development tools to refer to any number of features that make programming easier. [Archetypes](#) and [ArchGenXML](#) are examples of these from the Plone universe.

Request Each page view by a client generates a request to Plone. This incoming request is encapsulated in a *request* object in Zope, usually called REQUEST (or lowercase “request” in the case of ZPT).

ResourceRegistries A piece of Plone infrastructure that allows CSS/JavaScript declarations to be contained in separate, logical files before ultimately being appended to the existing Plone CSS/JavaScript files on page delivery. Primarily enables Product authors to “register” new CSS/JavaScript without needing to touch Plone’s templates, but also allows for selective inclusion of CSS/JavaScript files and reduces page load by minimizing individual calls to separate blocks of CSS/JavaScript files. Found in the [ZMI](#) under `portal_css` and `portal_javascript`.

reStructuredText The standard plaintext markup language used for Python documentation: <http://docutils.sourceforge.net/rst.html>

[reStructuredText](#) is an easy-to-read plaintext markup syntax and parser system. It is useful for in-line program documentation (such as Python docstrings), for quickly creating simple web pages, and for standalone documents. `reStructuredText` is designed to be extensible for specific application domains. The `reStructuredText` parser is a component of [Docutils](#).

reStructuredText is a revision and reinterpretation of the *StructuredText* and *Setext* lightweight markup systems.

Skin A collection of template layers (see *layer*) is used as the search path when a page is rendered and the different parts look up template fragments. Skins are defined in the *ZMI* in `portal_skins` tool. Used for both presentation and code customizations.

slug A *ZCML* *slug* is a one-line file created in a Zope instance's `etc/package-includes` directory, with a name like `my.package-configure.zcml`. The contents of the file would be something like:

```
<include package="my.package" file="configure.zcml" />
```

This is the Zope 3 way to load a particular package.

Software home The directory inside the Zope installation (on the filesystem) that contains all the Python code that makes up the core of the Zope application server. The various Zope packages are distributed here. Also referred to as the `SOFTWARE_HOME` environment variable. It varies from one system to the next, depending where you or your packaging system installed Zope. You can find the value of this in the *ZMI* > *Control Panel*.

Sprint Based on ideas from the extreme programming (XP) community. A sprint is a three to five day focused development session, in which developers pair in a room and focus on building a particular subsystem. See <https://plone.org/events/sprints>

STX

StructuredText Structured Text is a simple markup technique that is useful when you don't want to resort to HTML for creating web content. It uses indenting for structure, and other markup for formatting. It has been superseded by *reStructuredText*, but some people still prefer the old version, as it's simpler.

Syndication Syndication shows you the several most recently updated objects in a folder in RSS format. This format is designed to be read by other programs.

TAL Template Attribute Language. See *ZPT*.

TALES *TAL* Expression Syntax. The syntax of the expressions used in TAL attributes.

TinyMCE A graphical HTML editor bundled with Plone.

TODO The TODO marker in source code records new features, non-critical optimization notes, design changes, etc.

toolbar Plone uses a toolbar to have quick access to the content management functions. On a standard instance, this will appear on the left of your screen. However, your site administrator might change this to have a horizontal layout, and it will appear hidden at first when using a smaller-screen device like a phone or tablet.

Traceback A Python “traceback” is a detailed error message generated when an error occurs in executing Python code. Since Plone, running atop Zope, is a Python application, most Plone errors will generate a Python traceback. If you are filing an issue report regarding a Plone or Plone-product error, you should try to include a traceback log entry with the report.

To find the traceback, check your `event.log` log file. Alternatively, use the *ZMI* to check the `error_log` object in your Plone folder. Note that your Zope must be running in *debug* mode in order to log tracebacks.

A traceback will be included with nearly all error entries. A traceback will look something like this: “Traceback (innermost last): ... AttributeError: adapters” They can be very long. The most useful information is generally at the end.

traversal Publishing an object from the ZODB by traversing its parent objects, resolving security and names in scope. See the *Acquisition chapter in the Zope 2 book*. <http://docs.zope.org/zope2/zope2book/Acquisition.html>

TTP Actions done TTP are performed “Through the Plone” interface. It is normally a lazy way of telling you that you should not add things from the *ZMI*, as is the case for adding content, for example.

TTW This is a general term meaning an action can be performed “Through The Web,” as opposed to, say, being done programmatically.

UML The *Unified Modeling Language* is a general-purpose modeling language that includes a standardized graphical notation used to create an abstract model of a system, referred to as a *UML model*. With the use of [ArchGenXML](#), this can be used to generate code for CMF/Plone applications (a *Product*) based on the Archetypes framework.

virtualenv `virtualenv` is a tool for creating a project directory with a Python interpreter that is isolated from the rest of the system. Modules that you install in such an environment remain local to it, and do not impact your system Python or other projects.

VirtualHostMonster A Zope technology that supports virtual hosting. See [VirtualHostMonster URL rewriting mechanism](#)

Workflow Workflow is a very powerful way of mimicking business processes - it is also the way security settings are handled in Plone.

XXX XXX is a marker in the comments of the source code that should only be used during development to note things that need to be taken care of before a final (trunk) commit. Ideally, one should not expect to see XXXs in released software. XXX shall not be used to record new features, non-critical optimization, design changes, etc. If you want to record things like that, use TODO comments instead. People making a release shouldn't care about TODOs, but they ought to be annoyed to find XXXs.

ZCA The Zope Component Architecture (ZCA) is a Python framework for supporting component-based design and programming. It is very well suited to developing large Python software systems. The ZCA is not specific to the Zope web application server: it can be used for developing any Python application. From [A Comprehensive Guide to Zope Component Architecture](#).

ZCML Zope Configuration Markup Language. Zope 3 separates policy from the actual code and moves it out to separate configuration files, typically a `configure.zcml` file in a buildout. This file configures the Zope instance. 'Configuration' might be a bit misleading here and should be thought of more as wiring. ZCML, the XML-based configuration language that is used for this, is tailored to do component registration and security declarations, for the most part. By enabling or disabling certain components in ZCML, you can configure certain policies of the overall application. In Zope 2, enabling and disabling components means to drop in or remove a certain Zope 2 product. When it's there, it's automatically imported and loaded. This is not the case in Zope 3. If you don't enable it explicitly, it will not be found.

ZEO server ZEO (Zope Enterprise Objects) is a scaling solution used with Zope. The ZEO server is a storage server that allows multiple Zope instances, called ZEO clients, to connect to a single database. ZEO clients may be distributed across multiple machines. For additional info, see [the related chapter in The Zope Book](#).

ZMI The *Management Interface*. A Management Interface that is accessible through the web. Accessing it is as simple as appending `/manage` to your URL, for example: `http://localhost/manage` - or visiting Plone Setup and clicking the *Management Interface* link (Click 'View' to go back to the Plone site). Be careful in there, though - it's the "geek view" of things, and is not straightforward, nor does it protect you from doing stupid things. :)

ZODB The Zope Object Database is where your content is normally stored when you are using Plone. The default storage backend of the ZODB is *filestorage*, which stores the database on the file system in the file(s) such as `Data.fs`, normally located in the `var` directory.

Zope instance An operating system process that handles HTTP interaction with a Zope database (*ZODB*). In other words, the Zope web server process. Alternatively, the Python code and other configuration files necessary for running this process.

One Zope installation can support multiple instances. Use the buildout recipe `plone.recipe.zope2instance` to create new Zope instances in a buildout environment.

Several Zope instances may serve data from a single ZODB using a ZEO server on the back-end.

Zope product A special kind of Python package used to extend Zope. In old versions of Zope, all products were directories inside the special *Products* directory of a Zope instance; these would have a Python module name

beginning with `Products`. For example, the core of Plone is a product called *CMFPlone*, known in Python as `Products.CMFPlone`.

ZPL Zope Public License, a BSD-style license that Zope is licensed under.

ZPT *Zope Page Templates* is the templating language that is used to render the Plone pages. It is implemented as two XML namespaces, making it possible to create templates that look like normal HTML/XML to editors. See <http://docs.zope.org/zope2/zope2book/AppendixC.html>

Error Reference

older manuals

Some of these are still valid, and can give a deeper understanding of the Plone/Zope ecosystem.

Note: These manuals are not updated anymore

- [Buildout](#)
- [using zope.formlib](#)
- [using zope.formlib](#)
- [zope.formlib](#)
- [Zope 2 vs. Zope 3 practices](#)
- [Pluggable Authentication Service](#)
- [Portlets](#)
- [PloneTestCase tests](#)
- [Zope 2 internals](#)

About this documentation

Writing Documentation

Description

How to write and submit content for the Plone Documentation.

Reaching The Documentation Team

The Plone community runs a documentation team which is responsible for keeping the Plone documentation coherent. To reach this team for any questions please contact

- [Documentation](#) category on [community.plone.org](#).

For news and updates you can also follow [PloneDocs](#) on twitter.

Feedback

We need your feedback to make the documentation better !

If you have already a GitHub account, please do not hesitate to open a ticket in our [issue tracker](#) .

If you do not have one, please use the ‘Feedback’ widget on [docs.plone.org](#) .

is version, so if you are using Plone 4 consult the [Plone 4.3 Documentation](#)

Plone - The Open Source CMS

This is a community-maintained manual for the [Plone](#) content management system.

This documentation is for:

- Content editors: who write, update, and order content on a site
- Site administrators: who install Plone and add-ons, and set up a site
- Designers: who create site themes
- Deployers: who configure server(s) for site hosting
- Developers: who customize a site's capabilities, create add-ons, and contribute to Plone itself

Reporting Issues

Documentation Issues

Documentation issues are tracked in our [issue tracker](#).

If you have already a [GitHub](#) account and you find something missing or wrong in the docs, please open a ticket.

One of the best ways to start contributing is to pick one of the tickets labeled [First Timer](#).

Note: Working on issues takes time and energy, to make it as pleasant as possible for everyone, meaning for you and the reviewer we would like to ask you kindly to make yourself familiar with our style-guides for *documentation* and *reStructuredText [reST]*.

Plone Issues

If you think your bug involves a core component of Plone, check to see if that component has its own repository under the [Plone organization on GitHub](#).

If it does, use the component's issue tracker to submit your bug report. If the component does not have its own repository there, submit your bug report in [the catch-all CMFPlone tracker on GitHub](#).

Note: after figuring out where to post your report, but before doing so, please search the issue tracker to ensure this issue hasn't already been reported and that it hasn't already been fixed in a later version of the component.

You can also find a [overview](#) about that on [plone.org](#).

License

The Plone Documentation by [Plone Foundation](#) is licensed under a [Creative Commons Attribution 4.0 International License](#).

If you want to contribute to this documentation, you can do so directly by making a pull request, if you have filled out a [Contributor Agreement](#).

If you haven't filled in a Contributor Agreement, you can still contribute. Contact the Documentation team, for instance via our [community space](#).

Basically, all we need is your written confirmation that you are agreeing your contribution can be under Creative Commons.

You can also add in a comment with your pull request “I, <full name>, agree to have this published under Creative Commons 4.0 International BY”.

Workflow

The documentation is hosted on GitHub. And there are tools hooked directly into it:

- there are branches for the different versions of Plone, see [Documentation For Different Versions Of Plone](#).
- translation hooks with [Transifex](#) are in place.
- some external documentation is pulled in, to collect all the documentation in one place.

For these reasons, it is important we keep the documentation coherent. Therefore, we follow a simple workflow, which we ask all contributors to respect:

Please **DO NOT** commit to master directly. Even for the smallest and most trivial fix.

ALWAYS open a pull request and ask somebody else to merge your contribution. **NEVER** merge it yourself.

Your pull requests may be checked for spelling, and clarity. Don’t hesitate to contribute also if English is not your first language, we will try to be helpful in corrections without being annoying.

If you don’t get feedback on your pull request in a day please come to #plone-docs and ask.

The main goal of this process is not to annoy you. On the contrary, we **love** your contributions.

But the documentation team also wants to keep the documentation in good shape.

Documentation For Different Versions Of Plone

The documentation for the different versions (Plone 3, Plone 4, Plone 5) are organized in branches inside the [Plone Documentation](#)

The *default* branch points to the current version of Plone.

Documentation changes that are valid for multiple versions of Plone can be done by making multiple pull requests, or by cherry-picking which may be easier to do when branches are widely different.

When all this seems complicated, note in your pull request that you think this is valid for other versions of Plone as well, and the documentation team will take care.

Editing The Documentation On GitHub

This is the recommended way for smaller changes, and for people who are not familiar with Git.

- Go to [Plone Documentation](#) on GitHub.
- Press the *Fork* button. This will create your own personal copy of the documentation.
- **Edit** files using GitHub’s text editor in your web browser
- Fill in the *Commit changes*-textbox at the end of the page telling why you did the changes. Press the *Commit changes*-button next to it when done.
- Then head to the green *New pull request*-button (e.g. by navigating to your fork’s root and clicking *Pull requests* on the right menu-bar, or directly via <https://github.com/yourGitHubUserName/documentation/pulls>), you won’t need to fill in any additional text. Press *New pull request*-button, finally click *Send pull request*.
- Your changes are now queued for review under project’s [Pull requests](#) tab on GitHub.

- For more information about writing documentation please read the [styleguide](#) and also [this](#).
- You will receive a message when your request has been integrated into the documentation. At that moment, feel free to delete the copy of the documentation you created under your account on GitHub. Next time you contribute, just fork again. That way you'll always have a fresh copy of the documentation to work on.

Before You Make A Pull Request

- Check for typos. Again, do not let this discourage you if English is not your first language, but simple typing errors can usually be found with spellcheckers
- Make sure that all links you put in are valid.
- Check that you are using valid restructured text.

Pull Request Checklist

Making a good pull request makes life easier for everybody:

- The title and description of a pull request **MUST** be descriptive and need to reflect the changes. So please say “grammar fixes on the intro page” or “new page: feature x explained as a user story”

If you can state for which versions of Plone your submissions are valid, that would be awesome.

We use a template which creates a default form for pull requests

Fixes #.

Improves:

Changes proposed in this pull request:

You can view, comment on, or merge this pull request online at:

If possible please make sure to fill in the missing bits, for example

```
Fixes #1234

Improves:

- Style-guide about reST syntax

Changes proposed in this pull request: Unified usage of '..code-block:: shell' as
↪best practices
```

Editing The Documentation Using Git

This is the recommended method of editing the documentation for advanced users.

- Learn about [Sphinx](#) and [restructured text](#).
- [Fork](#) the documentation source files into your own repository
- Edit the file(s) which you want to update.
- Check that you do not have any syntax errors or typos
- Commit your changes and [create](#) and open [pull](#) request.

For more information about writing documentation please read the *styleguide* and also *this*.

Translation

We use [Transifex](#) for translation.

Quick Start:

- Browse to: <https://www.transifex.com/projects/p/plone-doc/> and choose your language.
- Click on the right *Join Team*

Getting Started

- Go to: <https://www.transifex.com/signin/>
- Go to: <https://www.transifex.com/projects/p/plone-doc/>
- Click on: *HELP TRANSLATE PLONE DOCUMENTATION*
- Choose your language
- Click on the right *Join Team*

Documentation Styleguide

Description

A guide to write Documentation for Plone and for Plone Add-ons

Introduction

Having a ‘best practices’ approach for writing your documentation will benefit the users and the community at large.

Even better: when there is a clear structure and style for your documentation, the chances that other people will help improve the documentation increase!

Further advantages of following this guide:

- The documentation can be included on docs.plone.org
- It will be in optimal format to be translated with tools like [Transifex](#).

Tone

Guides should be informational, but friendly. Use the active voice whenever possible, and contractions and pronouns are acceptable (in particular, the use of you in regards to the reader).

Use common sense – if a term is related to a high-level concept that fewer people would know, then take a sentence or two to explain it.

Your documentation is not code.

It needs to be translatable. No, not into PHP, but into Chinese, Catalan, Klingon, ...

Think about it this way:

Each sentence in the documentation can be turned into a .po string. Breaking sentences with linebreaks would mean a translator will only see part of the sentence, making it impossible to translate.

Documentation Structure & Styleguide

For including documentation into docs.plone.org, **please** follow these guidelines:

- All documentation should be written in **valid ReStructuredText** There are some *Helper tools for writing Documentation* available.
- The top level of your package should contain the following documentation-related files:
 - README.rst This should be a **short** description of your add-on, not the entire documentation! See the *README Example*
 - CHANGES.rst This should track the feature changes in your add-on, see *Tracking Changes*
 - CONTRIBUTORS.rst This should list the people writing, translating and otherwise contributing
- All of your (longer) end-user documentation should go into the /docs subdirectory. Feel free to split your documentation into separate files, or even further subdirectories if it helps clarity.
- Make **absolutely** sure there is a start page called index.rst. It is also usually a *really* good idea to have that include a Table of Content, see *Table Of Contents For Your Documentation*
- use relative links for internal links within your /docs/ directory, to include images for instance.
- If you want to include images and screenshots, you should place them into /docs/resources/ , along with other resources like PDF's, audio, video, etcetera. (*Yes! Make more screenshots, we love you! But do remember, .png or .jpg as file formats, no .gif please*)
- Please do not symlink to, or use the *include* directive on files that live outside your '/docs' directory.
- Please do not use 'autodoc' to include comments of your code.
- The '/docs' directory should contain **only** content related to documentation, please do **not** put the license here. A LICENSE.rst with a short description of the license, and LICENSE.GPL for the legalese should go into the top level of your package next to your README.rst
- Please follow this *ReStructuredText Style Guide* and use **semantic linefeeds**. Do **not** break your sentences into half with newlines because you somehow think you should follow PEP8. PEP8 is for Python files, not for ReStructuredText.
- Please follow our *Word choice*.
- Usage of *Sphinx* within your project is optional, but if you want your add-on to (also) be documented for instance on *Read The Docs* it is highly recommended. Put the associated Makefile and conf.py into the /docs directory.

Note: If you use *bobtemplates.plone* to generate the layout of your add-on, the recommended files will already be there, and in the right place. You'll still have to write the content, though.

README Example

This is an example of how a README.rst should look like:

```
=====
collective.fancystuff
=====
```

collective.fancystuff will make your Plone site more fancy.
It can do cool things, and will make the task of keeping your site fancy a lot easier.

The main audience for this are people who run a chocolate factory.
But it also is useful for organisations planning on world domination.

Features

```
=====
```

- Be awesome
 - Make things fancier
 - Works out of the box, but can also be customized.
- After installation, you will find a new item in your site control panel where to ↵
↵set various options.

Examples

```
=====
```

This add-on can be seen in action at the following sites:

- <http://fancysite.com>
- <http://fluffystuff.org>

Documentation

```
=====
```

Full documentation for end users can be found in the "docs" folder.
It is also available online at <http://docs.plone.org/foo/bar>

Translations

```
=====
```

This product has been translated into

- German

Installation

```
=====
```

Install collective.fancystuff by adding it to your buildout:

```
[buildout]

...

eggs =
    collective.fancystuff
```

and then running "bin/buildout"

Contribute

=====

- Issue Tracker: <https://github.com/collective/collective.fancystuff/issues>
- Source Code: <https://github.com/collective/collective.fancystuff>
- Documentation: <https://docs.plone.org/foo/bar>

Support

=====

If you are having issues, please let us know.
We have a mailing list located at: project@example.com

License

=====

The project is licensed under the GPLv2.

Tracking Changes

Feature-level changes to code are tracked inside `CHANGES.rst`. The title of the `CHANGES.rst` file should be Changelog. Example:

Changelog

=====

1.0.0-dev (Unreleased)

- Added feature Z.
[github_userid1]
- Removed Y.
[github_userid2]

1.0.0-alpha.1 (yyyy-mm-dd)

- Fixed Bug X.
[github_userid1]

Add an entry every time you add/remove a feature, fix a bug, etc. on top of the current development changes block.

Table Of Contents For Your Documentation

Make sure all `.rst` files are referenced with a Table of Contents directive, like this example:

```
.. toctree::  
    :maxdepth: 2  
  
    quickstart
```

```
working_examples
absolutely_all_options_explained
how_to_contribute
```

(note: the files themselves will have an extension of .rst, but you don't specify that extension in the toctree directive)

ReStructuredText Style Guide

Description

How to write content for the Plone Documentation.

Introduction

This chapter explains the basics of editing, and updating to the *Plone Documentation*.

Note: All pages should be in ReStructured Text, and have a .rst extension. Images should be in .png, or .jpg format. Please, don't use .gif, because the PDF-generating software has issues with that.

Line Length & Translations

Documentation is not code. Repeat after us: **Documentation is not code.**

Documentation should **not** follow [PEP8](#) or other arbitrary conventions.

Note: Remember : This documentation is set up so it is fully translatable by using standard tools like transifex.

Your sentences will become .po strings, to be translated.

Now, think about how translations would work if the translator can only see an arbitrary part of a sentence. Translating is hard enough without creating additional problems...

If you want to keep short lines:

Use [semantic linefeeds](#) when you are editing restructured text (or any other interpreted rich text format) because it will greatly improve the editing and maintenance of your documents.

Take this example paragraph:

```
Patterns can take options in two ways:
from the DOM or via the jQuery interface.
It is highly recommended to use the DOM interface,
since it offers a lot more flexibility compared to the jQuery approach.
Also,
if you wish to use the automatic binding and rebinding functionality,
the DOM approach is more straightforward and hassle-free.
```

Notice how it's easier to just reshuffle sentences and add stuff if, instead of using your editor "autowrap" feature, you manually insert line breaks after full stops, commas, or upon "grammatical" boundaries (and not merely word ones).

Do not be afraid to use more than 80 characters.

Document Page Format

Here are some Sphinx coding conventions used in the documentation.

Tab Policy

- Indentation 4 spaces
- No hard tabs
- No trailing whitespaces

Headings And Filenames

- For the headings, capitalize the first letter only
- For the filenames, use `_underscore_naming_style`

Page Structure

Each page should contain, in this order:

- The main heading. This will be visible in the table of contents:

```
=====
Writing and updating this document
=====
```

- The description of the page, which will appear in Plone's *Description* Dublin Core metadata field. This created using the reST *admonition* directive. A single paragraph of text consisting of 1-3 sentences is recommended, so that the same text fits into the search engine results (Google):

```
.. admonition:: Description
```

```
    This text will go to Plone's pages description field. It will appear in the search_
↪engine listings for the page.
```

Introduction paragraph: A brief overview:

```
Introduction
-----
```

```
This chapter will describe the basics of how to contribute to this document.
```

A number of paragraphs: The actual content of the document page:

```
Contributions needed
-----
```

```
Below is the list of documentation and references we'd like to see
```


Section Structure

Each section (folder) must contain

- `index.rst` with:
- Section heading: This will be visible in the table of contents
- A single paragraph summarizing what this section is all about. This will be mapped to Plone folder description.
- Sphinx `toctree` directive, `maxdepth 2`. Each `.rst` file in the folder should be linked to this toctree.

```
.. toctree::
   :maxdepth: 2

   chapter1
   chapter2
   chapter3
```

Headings

reStructuredText and Sphinx enable any style you would prefer for the various heading level you would need. For example, underlining level 1 headings with `..`, level 2 headings with `#` and level 3 headings with `|` is perfectly valid as far as `docutils` is concerned.

Unfortunately this is not the same for a human documentation maintainer.

For having consistent heading styles in all files it is recommended to follow strictly the rules stated in the [Sphinx manual](#).

As individual files do not have so called “parts” or “chapters”, the headings would be underlined like this:

```
===
One
===

Two
===

Three
-----

Four
~~~~

Five
^^^^
```

Links

Sphinx can use two link styles, inline and via a link at the end of the page. Please **do not** separate the link and the target definition, please **only** use inline links like this:

```
`Example <https://example.com>`_
```

otherwise the URL is not attached to the context it is used in, and that makes it harder for translators to use the right expressions.

Topic

A topic is like a block quote with a title, or a self-contained section with no subsections.

Use the “topic” directive to indicate a self-contained idea that is separate from the flow of the document. Topics may occur anywhere a section or transition may occur. Body elements and topics may not contain nested topics.

The directive’s sole argument is interpreted as the topic title; the next line must be blank.

All subsequent lines make up the topic body, interpreted as body elements. For example:

```
.. topic:: Topic Title

    Subsequent indented lines comprise
    the body of the topic, and are
    interpreted as body elements.
```

Syntax Highlighting

Sphinx does syntax highlighting using the [Pygments](#) library.

You can specify different highlighting for a code block using the following syntax:

With two colons you start a code block using the default highlighter::

```
# Some Python code here
# The language defaults to Python, we don't need to set it
if 1 == 2:
    pass
```

You can specify the language used for syntax highlighting by using the `code-block` directive:

```
.. code-block:: python

    if "foo" == "bar":
        # This is Python code
        pass
```

For example, to specify XML:

```
.. code-block:: xml

    <somesnippet>Some XML</somesnippet>
```

... or UNIX shell:

```
.. code-block:: shell

    # Start Plone in foreground mode for a test run
    cd ~/Plone/zinstance
    bin/plonectl fg
```

... or a buildout.cfg:

```
.. code-block:: ini

    [some-part]
    # A random part in the buildout
```

```
recipe = collective.recipe.foo
option = value
```

... or interactive Python:

```
.. code-block:: pycon

>>> class Foo:
...     bar = 100
...
>>> f = Foo()
>>> f.bar
100
>>> f.bar / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

... or JavaScript:

```
.. code-block:: javascript

var $el = $('<div/>');
var value = '<script>alert("hi")</script>';
$el.text(value);
$('body').append($el);
```

Setting the highlighting mode for the whole document:

```
.. highlight:: shell

All code blocks in this doc use console highlighting by default::

    some shell commands
```

If syntax highlighting is not enabled for your code block, you probably have a syntax error and `Pygments` will fail silently.

Images

reST supports an image directive:

```
.. image:: ../_static/plone_donut.png
:alt: Picture of Plone Donut
```

When used within Sphinx, the file name given (here `plone_donut.png`) must either be relative to the source file, or absolute which means that they are relative to the top source directory.

For example, the file `sketch/spam.rst` could refer to the image `_static/plone_donut.png` as `../_static/plone_donut.png` or `/_static/plone_donut.png`.

Other Sphinx And ReStructured Text Source Snippets

Italics:

```
This *word* is italics.
```

Strong:

```
This **word** is in bold text.
```

Inline code highlighting:

```
This is :func:`aFunction`, this is the :mod:`some.module` that contains the ↪:class:`some.module.MyClass`
```

Note: These Python objects are rendered as hyperlinks if the symbol is mentioned in a relevant directive. See <http://sphinx-doc.org/domains.html> and <http://sphinx-doc.org/ext/autodoc.html>

Making an external link (note the underscore at the end):

```
`This is an external link to <http://opensourcehacker.com>`_
```

Making an internal link:

```
:doc:`This is a link to </introduction/writing.txt>`  
...  
See also :ref:`somewhere` (assuming that a line containing only  
``.. _somewhere:`` exists above a heading in any file of this  
documentation) ...  
And a link to the term :term:`foo` assuming that ``foo`` is defined in the glossary.
```

Glossary:

```
.. glossary:: :sorted:
```

Bullet list:

```
* First bullet  
* Second bullet with `a link <http://opensourcehacker.com>`_
```

Warning:

```
.. warning::  
  
    This is a warning box (yellow)
```

Warning: This is a warning box (yellow)

```
.. error::  
  
    This is an error box (red)
```

Error: This is an error box (red)

Note:

```
.. note::

    This is a note box (blue)
```

Note: This is a note box (blue)

```
.. TODO::

    This is a TODO item
```

You can find a brief introduction to reStructuredText (reST) on <http://www.sphinx-doc.org/en/stable/rest.html>

Including Gists

Sometimes it is handy to include [gists](#). This can be useful if you want to include for example a configuration file.

For including gists just use the *gist* directive

```
.. gist:: https://gist.github.com/shomah4a/5149412
```

Note: Since this documentation serves as source for various versions (html, PDF, others), please **always** include a link to the gist under the gist directive.

Word choice

The following table lists the preferred word or phrase choice.

Preferred word choice

Word	Do not use
and so on	etc.
delete	destroy
plug-in	plugin
add-on	addon
select	check
such as, as in	like
verify	be sure
GitHub	Github, github
JavaScript	Javascript, javascript
jQuery	jquery
Management Interface	ZMI

Helper tools for writing Documentation

Description

Tools and Plugins which will help to write documentation.

Online tools

- rst.ninjs.org and a fork with more Sphinx support at livesphinx.herokuapp.com
- notex.ch

Offline tools

ReText is a handy editor for **.rst** and **.md** formats. On Ubuntu and Debian-based systems all you have to do is

```
apt-get install retext
```

Pandoc If you have existing documentation, you may want to check out [pandoc](http://pandoc.org), the “swiss army knife” of document conversions. For instance, it can create valid rst files from Markdown and many other formats. On Ubuntu you can install it via apt

```
apt-get install pandoc
```

See also the [online version](#).

Sublime Text has a number of plugins for rst highlighting and snippets, install via the Sublime package installer.

Very useful is [Restructured Text Snippets](#), which has automated header creation, html preview and more. It even makes creating tables in ReST tolerable. **Highly recommended.**

The *SublimeLinter* framework also comes with two plugins: [sublimelinter-rst](#) will error check RST files, and [write good](#) checks your English for writing style.

When not using the Snippets, but you still want to check how the html would look, [OmniMarkupPreviewer](#) is a live previewer/exporter for markup files (markdown, rst, creole, textile...).

Watch [the video](#) on youtube.

Emacs has a nice [rst-mode](#). This mode comes with some Emacs distros. Try `M-x rst-mode` in your Emacs and enjoy syntax coloration, underlining a heading with `^C ^A`

Another nice tool for Emacs is [Flycheck](#).

Eclipse users can install **ReST Editor** through the Eclipse Marketplace.

Vim does syntax highlighting for RST files. There is a nice plugin called [vim-markdown](#).

If you prefer a more *advanced* plugin with enhanced functionalities you could use [Riv](#).

Restview [ReStructuredText viewer](#) A viewer for ReStructuredText documents that renders them on the fly.

```
pip install restview
```

Language tools

These tools can help you to check for grammatical mistakes and typos, you should always use a spell checker anyway!

LanguageTool is an Open Source proofreading software for English, French, German, Polish and more than 20 other languages. See www.languagetool.org.

After the Deadline is a [language checker for the web](#). This handy tool is also available in your Plone sites, by the way, see the [content quality](#) section

Searching the documentation

Description

Tools for searching in the documentation.

Firefox

Our [documentation](#) is supporting [OpenSearch](#).

With this it is possible to search [docs.plone.org](#) with put the Firefox search bar.

Watch the [video](#) .

License for Plone Documentation

Plone Documentation by Plone Foundation is licensed under a [Creative Commons Attribution 4.0 International License](#).

If you want to contribute to this documentation, you can do so directly by making a pull request, if you have filled out a [Contributor Agreement](#).

If you haven't filled in a Contributor Agreement, you can still contribute. Contact the Documentation team, for instance via the [mailinglist](#) or directly send a mail to plone-docs@lists.sourceforge.net

Guidelines and Examples

If you need help with an error or problem: before asking the question, please take a few minutes to read the guidelines below. It is important to know how to state questions, because once you learn it, you will get better answers back more quickly.

Asking help on support and discussion forums

By stating a **well-phrased question** you increase the likelihood of fast and helpful responses to your question.

Here are some general key rules users need to follow in creating a new topic.

- ALWAYS start with searching before you ask a question. Most of your questions were probably already answered by someone else in the past. Save your and our time by searching the web first.

Where to search:

- Google - Before asking for help, make a Google search with related keywords. Pick meaningful keywords from the log entry. Sometimes searching for the entire error message works!
- [Plone Community Forums](#) - help and discussion forums
- [StackOverflow](#) - some FAQs are maintained here
- [Troubleshooting](#) tips and common error messages - for enabling debug mode and common tracebacks
- [GitHub issue tracker](#) - for known related issues
- [Documentation issue tracker](#) - for documentation related issues

If at any point you see any kind of error message (including error codes) – put them in your question. Never write anything like “I see some error message”. Be specific.

See [Basic troubleshooting](#) in case of an error

Follow netiquette while visiting and writing on forum or mailing list (give respect = get respect). This includes:

- Be patient – sometimes the problem cannot be solved within minutes or hours. You might need to bump the topic few times until an experienced person comes to the site and sees it, but usually if you do not see any response after a day or two it probably means we do not know the answer to your question, or perhaps your question needs to include more detail.
- Do not use bad words. Respect others and what they are doing.
- Do not completely edit/erase your posts after you posted them on the forum (except for small corrections - they are allowed) Remember that once you sent them, they belong to the community and shall be used by anyone who needs it.

How to write a good topic

Keep in mind, that if you ask a question and all you hear is silence, it might be a good indicator that something is wrong with your topic. Read the hints below and try to match your topic with specified pattern.

Subject lines: most people will read a message only if it appears to be intelligent. Your subject line is your sales pitch, so you should make your subject line specific and easy to understand.

Note: A poor subject line:

GET METHOD!! URGENT HELP!!!!

A better subject line:

FooError in Passing GET variables to FormController

The big picture - An opening sentence should state the general problem that you wish to solve.

A snapshot of your environment - For Plone and for other relevant products: provide version numbers. e.g., “I’m running Plone 5.0.4 under Python 2.7.11.”

Steps to reproduce the issue - Give information about your ideas of how this error appeared, what caused it or anything that could lead to reproducing the error on another computer, including your buildout.cfg, versions.cfg, version numbers of installed add-ons, detailed command lines, the complete error message stack. Mention your expected result.

Asking for help in online chat

The Plone community uses [online chat](#), specifically [Gitter](#) and traditional Plone IRC chat at the #plone IRC channel on the freenode IRC network.

(To use Gitter you must log in using a free GitHub or Twitter account, which allows you to receive notifications when there is a response to your question. IRC does not require you to log in, but you will receive notifications of responses only while you remain connected).

Here are useful hints for using online chat:

- Remember that chat participants are volunteers; they are not paid to provide support.
- Do not ask permission to ask a question, but directly start the conversation having the all necessary input. Follow the example below:

Hi! I am trying to install PloneFormGen product, but it does not appear in the add on products list.

When I start Zope in debug mode I get the following log entry.

I pasted the log to pastie.org and here is the link for the log entry <http://pastie.org/123123>

- Be specific - tell us why you are trying to accomplish something and then tell us what the problem is. Here are some guidelines how to form a good question for Internet discussion.
- Do not copy-paste text to chat. This disrupts other people chatting about other topics. Instead please paste the full traceback error log to pastie.org and then paste the link to your error log or code (from your browser's address bar) to the chat.
- Do not send direct messages to chat participants unless you have a clear reason to do so
- Keep the chat window open at least 30 minutes so that someone has time to pick up your question. Be patient.
- Do not repeat yourself - people might be busy or not able to help with your problem. Silence doesn't mean we're ignoring you, it means that nobody is online right now who knows the answer to your question.
- Do not overuse CAPS-LOCK writing, since it is considered shouting and nobody likes when others shout at them. Do not use excessive exclamation marks (!!!) or question marks (???) as it makes you look unprofessional and discourages to help you.
- There are many people discussing simultaneously - if you address a message to a particular person, use his or her nick name. Hint: you can use Tab key to autocomplete nick name after typing few letters.
- Chat is a real-time communication tool. Keep in mind, that since you write something, and send it, it cannot be taken back.
- Try to respond to all questions other users have. Chat is much more fluid and dynamic than the forum, so don't worry if you forget about putting something in the first message – you can still keep up.
- Do not worry if you are not fluent in English - Plone is a global community, and people will usually try to ask you more detailed questions in a way that the message gets through.

Note: Examples

An ineffective chat question:

“Anyone here using product XYZ? Anyone here have problems installing XYZ?”

A question that is more likely to gain attention and a positive response:

“Hi, I'm using product XYZ on Plone 5.x.x, I have a problem with the feature that is supposed to doABC— I get error BlahBlahError — what might be wrong? Here is a link to the error log on pastie.org:<http://pastie.org/123123>”

Tracebacks

When there is an error, a Python program always produces a traceback, a complete information where the application was when the error happened. To help you with an error, a complete traceback log is needed, not just the last line which says something like “AttributeError”.

Copy full tracebacks to your message (discussion forums) or pastie.org link (chat). The most reliable way to get the traceback output is to start Plone (Zope application server) in foreground mode in your terminal / command line (see these [debugging tips](#))

First, shut down Plone if it's running as a service / background process. Then start Plone in foreground mode.

On Linux, OSX or similar systems this is (navigate to Plone folder first):

```
bin/instance fg
```

On Windows command prompt this is

```
cd "C:\Program Files\Plone"  
bin\buildout.exe fg
```

Zope outputs all debug information to the console where it was started in foreground mode. When the error happens, the full traceback is printed to the console as well.

If Zope does not start in foreground mode it means that your add-on configuration is bad and you need to fix it and the related traceback is printed as well. In production mode, Zope ignores all add-ons which fail to load.

Credits

This how-to originated as an informal, user-friendly alternative to Eric Raymond's [How to Ask Questions the Smart Way](#). ESR's doc is long and offensive, though once you realize that ESR is your crusty old merchant-marine uncle it can be fun and helpful.

The error report format is adapted from Joel Spolsky's comments on bug tracking, e.g., in [Joel on Software](#).

Symbols

.po, [1091](#)

A

Acquisition, [1091](#)
adapter, [1005](#)
AGX, [1091](#)
annotation, [1003](#)
Archetypes, [1091](#)
ArchGenXML, [1091](#)
ATCT, [1091](#)

B

BBB, [1091](#)
browserview, [1091](#)
Buildout, [1092](#)

C

Catalog, [1092](#)
CMF, [1092](#)
Collective, [1092](#)
control panel, [1092](#)
CSS, [1092](#)

D

Dexterity, [1092](#)
Diazo, [1092](#)
Document, [1092](#)
DTML, [1092](#)
Dublin Core, [1092](#)

E

easy_install, [1092](#)
Egg, [1092](#)
Expiration Date, [1092](#)

G

GenericSetup, [1092](#)
gettext, [1092](#)

I

i18n, [1093](#)
i18ndude, [1093](#)

J

JSON, [1093](#)

K

KSS, [1093](#)
Kupu, [1093](#)

L

l10n, [1093](#)
Layer, [1093](#)
LDAP, [1093](#)

M

Manager, [1093](#)
METAL, [1093](#)
Monkey patch, [1093](#)

N

Namespace package, [1093](#)

O

OpenID, [1093](#)

P

PAS, [1093](#)
PLIP, [1094](#)
Plonista, [1094](#)
Product, [1094](#)
Python egg, [1094](#)
Python package, [1094](#)
Python Package Index, [1094](#)
Python path, [1094](#)

R

RAD, [1094](#)

Request, [1094](#)
ResourceRegistries, [1094](#)
reStructuredText, [1094](#)

S

Skin, [1095](#)
slug, [1095](#)
Software home, [1095](#)
Sprint, [1095](#)
StructuredText, [1095](#)
STX, [1095](#)
Syndication, [1095](#)

T

TAL, [1095](#)
TALES, [1095](#)
TinyMCE, [1095](#)
TODO, [1095](#)
toolbar, [1095](#)
Traceback, [1095](#)
traversal, [1095](#)
TTP, [1095](#)
TTW, [1095](#)

U

UML, [1096](#)

V

virtualenv, [1096](#)
VirtualHostMonster, [1096](#)

W

Workflow, [1096](#)

X

XXX, [1096](#)

Z

ZCA, [1096](#)
ZCML, [1096](#)
ZEO server, [1096](#)
ZMI, [1096](#)
ZODB, [1096](#)
Zope instance, [1096](#)
Zope product, [1096](#)
ZPL, [1097](#)
ZPT, [1097](#)