Plone Training Documentation

Release 1.2.5a

Plone Community

May 17, 2017

Contents

A collection of trainings developed and created by the Plone Community.

CHAPTER 1

About Plone Trainings

Plone Training is a collection of different trainings, developed and created by the Plone Community.

About Mastering Plone

This training was created by Philip Bauer and Patrick Gerken of starzel.de to create a canonical training for future Plone developers. The aim is that anyone with the appropriate knowledge can give a training based on it and contribute to it. It is published as Open Source on github and training.plone.org.

If you want to inquire the original authors about organizing a training please contact them at team@starzel.de.

Upcoming Trainings

If you want to have a training near you please ask for trainings on https://community.plone.org

Previous Trainings

The Mastering Plone Training was so far held publicly at the following occasions:

- Ploneconf 2016 in Boston
- October 2015, Bucharest
- March 2015, Munich
- Plone Conference 2014, Bristol
- June 2014, Caracas
- May 2014, Munich
- PythonBrasil/Plone Conference 2013, Brasilia
- PyCon DE 2012, Leipzig
- Plone Conference 2012, Arnheim
- PyCon De 2011, Leipzig

Trainers

The following trainers have given trainings based on Mastering Plone:

Philip Bauer Philip Bauer is a web developer from Munich who fell in love with Plone in 2005 and since then works almost exclusively with Plone. A historian by education he drifted towards creating websites in the 90's and founded the company Starzel.de in 2000. He is a member of the Plone foundation, loves teaching and is dedicated to Open Source. Among other Plone-related projects he started creating the Mastering Plone Training so that everyone can become a Plone-Developer.

Patrick Gerken Patrick Gerken works with Python since 2002. He started working with pure Zope applications and now develops mainly with Plone, Pyramid and Javascript as well as doing what is called DevOps. He works at Zumtobel Group.

Steve McMahon Steve McMahon is a long-time Plone community member, contributor and trainer. He is the creator of PloneFormGen and maintainer of the Unified installer. Steve also wrote several chapters of Practical Plone and is an experienced speaker and instructor.

Steffen Lindner Steffen Lindner started developing Plone in 2006. He worked on small Plone sites and also with huge intranet sites. As Open Source / Free Software developer he joined the Plone core developer team 2011 and works at Starzel.de.

Fulvio Casali Fulvio Casali has been working almost exclusively with Plone since 2008. He struggled for years to find his way around the source code of Plone when there was no documentation and no trainings, and feels passionate about helping users and developers become proficient. He loves participating in Plone community events, and organized two strategic Plone sprints on the northwest coast of the USA and helped galvanized the developer community there.

Johannes Raggam Johannes Raggam from Graz/Austria works most of the time with a technology stack based around Python, Plone, Pyramid and Javascript. As an active Open Source / Free Software developer he believes in the power of collaborative work. He is a BlueDynamics Alliance Partner and Plone Core Contributor since 2009, a member of the Plone Framework Team since 2012 and Plone Foundation member.

Franco Pellegrini Franco Pellegrini is a software developer from Cordoba, Argentina. He started developing Plone in 2005 in a small software company, and as an independent contractor since 2011. He believes in free software philosophy, and so, he has been a Plone core developer since 2010 and Framework Team member since 2012.

Fred van Dijk Fred, from Rotterdam the Netherlands, has been exposed to Plone early on as a user. In 2007 he joined Zest Software to work on and with Plone and Python web apps full time. He can focus on the business side, helping users decide on which features are most valuable to develop or when to stick with standard functionality. He also gives training on using and administering the CMS. On the IT side he has plenty technical knowledge to work on code, system administration and do project management in a team of developers.

Leonardo Caballero Leonardo J. Caballero G. of Maracaibo, Venezuela, is a Technical Director at Covantec R.L. and Conectivo C.A. Leonardo maintains the Spanish translations of more than 49 Plone Add-ons as well as Spanish-language documentation for Plone itself. He has contributed several Plone Add-ons that are part of PloneGov. Currently serving the Plone Board as a Plone Ambassador, Leonardo has also served as an Advisory Board member and has spoken at or helped organize Plone and open-source events throughout South America.

Using the documentation for a training

Feel free to organize a training yourself. Please be so kind to contribute any bug fixes or enhancements you made to the documentation for your training.

The training is rendered using sphinx and builds in two flavors:

default The verbose version used for the online documentation and for the trainer. Build it in sphinx with `make html` or use the online version.

presentation A abbreviated version used for the projector during a training. It should use more bullet points than verbose text. Build it in sphinx with `make presentation`.

Note: By prefixing an indented block of text or code with `.. only:: presentation` you can control that this block is used for the presentation version only.

To hide a block from the presentation version use `.. only:: not presentation`

Content without a prefix will be included in both versions.

The readthedocs theme

We slightly tweaked readthedocs theme in `_static/custom.css` so that it works better with projectors:

- We start hiding the navigation bar much earlier so that it does not interfere with the text.
- We enlarge the default width of the content-area.

Exercises

Some additional javascript shows hidden solutions for exercises by clicking.

Just prepend the solution with this markup:

```
..  admonition:: Solution
    :class: toggle
```

Here is a full example:

```
Exercise 1
^^^^^^^^^^

Your mission, should you choose to accept it...

..  admonition:: Solution
    :class: toggle

    To save the world with only seconds to spare do the following:

    .. code-block:: python

        from plone import api
```

It will be rendered like this:

Exercise 1

Your mission, should you choose to accept it...

Solution

To save the world with only seconds to spare do the following:

```
from plone import api
```

Building the documentation locally

Dependencies

Please make sure that you have Enchant installed, this is needed for spell-checking.

Install Enchant on OS X:

```
brew install enchant
```

Install Enchant on Ubuntu:

```
sudo apt-get install enchant
```

To build the documentation follow these steps:

```
$ git clone https://github.com/plone/training.git --recursive
$ cd training
$ virtualenv --python=python2.7 .
$ source bin/activate
```

Now install dependencies and build.

```
$ pip install -r requirements.txt
$ make html
```

You can now open the output from `_build/html/index.html`. To build the presentation version use `make presentation` instead of `make html`. You can open the presentation at `presentation/index.html`.

Build new

```
$ git clone https://github.com/plone/training.git --recursive
$ cd training
$ virtualenv --python=python2.7 .
$ source bin/activate
$ pip install -r requirements.txt
$ make html
```

Now you can open documentation with your web-bowser.

If you use OS X you can just do:

```
$ open _build/html/index.html
```

In the case of Linux, Ubuntu for example you can do:

```
$ firefox _build/html/index.html
```

Note: If you do not use Firefox but Chrome, please replace firefox with google-chrome e.g

```
$ google-chrome _build/html/index.html
```

Update existing

```
$ git pull
$ source bin/activate
$ make html
$ open _build/html/index.html
```

Technical set up to do before a training (as a trainer)

- Prepare a mailserver for the user registration mail (See *Configure a Mailserver*)
- If you do only a part of the training (Advanced) prepare a database with the steps of the previous sections. Be aware that the file- and blobstorage in the Vagrant box is here: /home/vagrant/var/ (not at the buildout path /vagrant/buildout/)

Upgrade the vagrant and buildout to a new Plone-version

- In https://github.com/collective/training_buildout change buildout.cfg to extend from the new *versions.cfg* on http://dist.plone.org/release
- Check if we should to update any versions in https://github.com/collective/training_buildout/blob/master/versions.cfg
- Commit and push the changes to the training_buildout
- Modify the vagrant-setup by modifying `plone_training_config/manifests/plone.pp`. Set the new Plone-version as *$plone_version* in line 3.
- Test the vagrant-setup it by creating a new vagrant-box using the new config.
- Create a new zip-file of all files in *plone_training_config* and move it to *_static*:

  ```
  $ cd plone_training_config
  $ zip -r plone_training_config.zip *
  $ mv plone_training_config.zip ../_static/
  ```

- Commit and push the changes to https://github.com/plone/training

Train the trainer

If you are a trainer there is a special mini training about giving technical trainings. We really want this material to be used, re-used, expanded and improved by Plone trainers world wide. These chapters don't contain any Plone specific advice, there's background, theory, check lists and tips for anyone trying to teach technical subjects.

../trainthetrainer/index

Contributing

Everyone is **very welcome** to contribute. Minor bug fixes can be pushed directly in the repository, bigger changes should made as pull-requests and discussed previously in tickets.

License

The Mastering Plone Training is licensed under a Creative Commons Attribution 4.0 International License.

Make sure you have filled out a Contributor Agreement.

If you haven't filled out 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 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".

CHAPTER 2

Trainings

Mastering Plone Development Mastering Plone is a training intended for people who are new to Plone or want to learn about the best practices of Plone development. In the course of the training you will learn how to build a custom website with plenty of features. Html and python-knowledge is required.

"Through-the-web" Plone customization Create custom content types, a design for a website, layouts for homepages and content types, and custom application logic. All in the browser!

Mastering Plone Theming Create a Diazo-based theme as a Plone add-on.

Mastering Plone Workflow How to create and make optimum use of custom Plone workflows

JavaScript for Plone Developers Learn best practices in Javascript development, how to develop and test your own patterns, and how to integrate your custom Javascript applications with Plone. Technologies will include NPM, Grunt, Patternslib and React.

Automating Plone Deployment How to automate deployment of Plone servers, whether it's one server or 100.

Plone Training Solr How to add enterprise-grade search to your Plone site.

Mastering Plone Development

This is the documentation for the "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 can be split in two trainings:

- A beginner training (2 to 3 days) that covers chapters 1-18.
- An advanced training (3 to 5 days) that covers the rest.

At conferences a shortended 2-day version of the advanced training with a slightly modified order is held.

Contents:

Introduction

Who are you?

Tell us about yourselves:

- Name, company, country...
- What is your Plone experience?

- What is your web development experience?
- What are your expectations for this tutorial?
- What is your favorite text editor?
- **If this training will include the development chapters:**
    - Do you know the HTML of the output of this?

      ```
      <div class="hiddenStructure"
           tal:repeat="num python:range(1, 10, 5)"
           tal:content="structure num"
           tal:omit-tag="">
        This is some weird sh*t!
      </div>
      ```

      The answer is:

      ```
      1 6
      ```

    - Do you know what the following would return?:

      ```
      [(i.Title, i.getURL()) for i in context.getFolderContents()]
      ```

What will we do?

Some technologies and tools we use during the training:

- For the beginning training:
    - Virtualbox
    - Vagrant
    - Ubuntu linux
    - Through-the-web (TTW)
    - Buildout
    - A little XML
    - A little Python
- For the advanced chapters:
    - Git
    - GitHub
    - Try Git (Nice introduction to git and github)
    - TAL
    - METAL
    - ZCML
    - Python
    - Dexterity
    - Viewlets

  - JQuery
  - Testing
  - References/Relations

What will we not do?

We will not cover the following topics:

- Archetypes
- Portlets
- z3c.forms
- Theming
- i18n and locales
- Deployment, Hosting and Caching
- grok

Other topics are only covered lightly:

- Zope Component Architecture
- GenericSetup
- ZODB
- Security
- Permissions
- Performance and Caching

What to expect

At the end of the first two days of training, you'll know many of the tools required for Plone installation, integration and configuration. You'll be able to install add-on packages and will know something about the technologies underlying Plone and their histories.

At the end of the second two days, you won't be a complete professional Plone-programmer, but you will know some of the more powerful features of Plone and should be able to construct a more complex website with custom themes and packages. You should also be able to find out where to look for instructions to do tasks we did not cover. You will know most of the core technologies involved in Plone programming.

If you want to become a professional Plone developer or a highly sophisticated Plone integrator you should definitely read Martin Aspeli's book and then re-read it again while actually doing a complex project.

Classroom Protocol

Note:

- Stop us and ask questions when you have them!
- Tell us if we speak too fast, too slow or not loud enough.
- One of us is always there to help you if you are stuck. Please give us a sign if you are stuck.

- We'll take some breaks, the first one will be at XX.
- Where is food, restrooms
- Someone please record the time we take for each chapter (incl. title)
- Someone please write down errors
- Contact us after the training: team@starzel.de

Questions to ask:

- What did you just say?
- Please explain what we just did again?
- How did that work?
- Why didn't that work for me?
- Is that a typo?

Questions __not__ to ask:

- **Hypotheticals**: What happens if I do X?
- **Research**: Can Plone do Y?
- **Syllabus**: Are we going to cover Z in class?
- **Marketing questions**: please just don't.
- **Performance questions**: Is Plone fast enough?
- **Unpythonic**: Why doesn't Plone do it some other way?
- **Show off**: Look what I just did!

Documentation

Follow the training at https://training.plone.org/5

Note: You can use this presentation to copy & paste the code but you will memorize more if you type yourself.

Further Reading

- Martin Aspeli: Professional Plone4 Development
- Practical Plone
- Zope Page Templates Reference

Installation & Setup

Installing Plone

The following table shows the Python versions required by Plone from version 3.x to 5.0.x:

Plone	Python
3.x	2.4
4.0.x	2.6
4.1.x	2.6
4.2.x	2.6 or 2.7
4.3.x	2.7
5.0.x	2.7

(Hopefully you won't have to deal with any Plone sites older than version 3.x.)

Plone 5.x requires a working Python 2.7 and several other system tools that not every OS provides. Therefore the installation of Plone is different on every system. Here are some ways that Python can be used:

- use a Python that comes pre-installed in your operating system (most Linux Distributions and Mac OS X have one)
- use the python buildout
- building Linux packages
- homebrew (Mac OS X)
- PyWin32 (Windows)

Mac OS X 10.8 - 10.10 and Ubuntu 14.04 come with a working default Python 2.7 built in. These are the lucky ones.

Most developers use their primary system to develop Plone. For complex setups they often use Linux virtual machines.

- OS X: Use the python buildout to compile python and homebrew for some missing Linux tools.
- Linux: Depending on your Linux flavor you might have to build python yourself and install some tools.
- Windows: Alan Runyan (one of Plone's founders) uses it. A downside: Plone seems to be running much slower on Windows.

Plone offers multiple options for being installed:

1. Unified installers (all 'nix, including OS X)
2. A Vagrant/VirtualBox install kit (all platforms)
3. A VirtualBox Appliance
4. Use your own Buildout

You can download all of these at https://plone.org/download

For the training we'll use option 2 and 4 to install and run Plone. We'll create our own Buildout and extend it as we wish. But we will do so in a vagrant machine. For your own first experiments we recommend option 1 or 2 (if you have a Windows laptop or encounter problems). Later on you should be able to use your own Buildout (we'll cover that later on).

See also:

- http://docs.plone.org/manage/installing/installation.html

Hosting Plone

If you want to host a real live Plone site yourself then running it from your laptop is not a viable option.

You can host Plone...

- with one of many professional hosting providers
- on a virtual private server

- on dedicated servers
- on heroku you can run Plone for *free* using the Heroku buildpack for Plone
- in the cloud (e.g. using Amazon EC2 or Codio.com)

See also:

- Plone Installation Requirements: http://docs.plone.org/manage/installing/requirements.html
- Run Plone on a 5$ plan: https://www.stevemcmahon.com/steves-blog/plone-on-5-a-month
- Where to host Plone: https://old.plone.org/documentation/faq/where-can-i-host-my-plone-site

Production Deployment

The way we're setting up a Plone site during this class may be adequate for a small site — or even a very large one that's not very busy — but you're likely to want to do much more if you're using Plone for anything demanding.

- Using a production web server like Apache or Nginx for URL rewriting, SSL and combining multiple, best-of-breed solutions into a single web site.
- Reverse proxy caching with a tool like Varnish to improve site performance.
- Load balancing to make best use of multiple core CPUs and even multiple servers.
- Optimizing cache headers and Plone's internal caching schemes with plone.app.caching.

And, you'll need to learn strategies for efficient backup and log file rotation.

All these topics are introduced in Guide to deploying and installing Plone in production.

Installing Plone for the Training

Keep in mind that you need a fast internet connection during installation since you'll have to download a lot of data!

> **Warning:** If you feel the desire to try out both methods below (with Vagrant and without), make sure you use different `training` directories! The two installations do not coexist well.

Installing Plone without vagrant

> **Warning:** If you are **not** used to running Plone on your laptop skip this part and continue with *Install VirtualBox*.

If you **are** experienced with running Plone on your own laptop, we encourage you to do so because you will have certain benefits:

- You can use the editor you are used to.
- You can use *omelette* to have all the code of Plone at your fingertips.
- You do not have to switch between different operating systems during the training.

If you feel comfortable, please work on your own machine with your own Python. But **please** make sure that you have a system that will work, since we don't want you to lose valuable time!

Note: If you also want to follow the JavaScript training and install the JavaScript development tools, you need NodeJS installed on your development computer.

Note: Please make sure you have your system properly prepared and installed all necessary prerequisites. For example, on Ubuntu/Debian, you need to install the following:

```
sudo apt-get install python-setuptools python-virtualenv python-dev build-essential␣
↪libssl-dev libxml2-dev libxslt1-dev libbz2-dev libjpeg62-dev
sudo apt-get install libreadline-dev wv poppler-utils
sudo apt-get install git
```

For more information or in case of problems see the official installation instructions.

Set up Plone for the training like this if you use your own OS (Linux or Mac):

```
$ mkdir training
$ cd training
$ git clone https://github.com/collective/training_buildout.git buildout
$ cd buildout
$ virtualenv --python=python2.7 py27
```

Now you can run the buildout for the first time:

```
$ ./py27/bin/python bootstrap.py
$ ./bin/buildout
```

This will take some time and produce a lot of output because it downloads and configures Plone. Once it is done you can start your instance with

```
$ ./bin/instance fg
```

The output should be similar to:

```
2015-09-24 15:51:02 INFO ZServer HTTP server started at Thu Sep 24 15:51:02 2015
        Hostname: 0.0.0.0
        Port: 8080
2015-09-24 15:51:05 WARNING PrintingMailHost Hold on to your hats folks, I'm a-patchin
↪'
2015-09-24 15:51:05 WARNING PrintingMailHost

******************************************************************************

Monkey patching MailHosts to print e-mails to the terminal.

This is instead of sending them.

NO MAIL WILL BE SENT FROM ZOPE AT ALL!

Turn off debug mode or remove Products.PrintingMailHost from the eggs
or remove ENABLE_PRINTING_MAILHOST from the environment variables to
return to normal e-mail sending.

See https://pypi.python.org/pypi/Products.PrintingMailHost

******************************************************************************
```

```
2015-09-24 15:51:05 INFO ZODB.blob (54391) Blob directory `.../buildout/var/
↪blobstorage` is unused and has no layout marker set. Selected `bushy` layout.
2015-09-24 15:51:05 INFO ZODB.blob (54391) Blob temporary directory '.../buildout/var/
↪blobstorage/tmp' does not exist. Created new directory.
.../.buildout/eggs/plone.app.multilingual-3.0.11-py2.7.egg/plone/app/multilingual/
↪browser/migrator.py:11: DeprecationWarning: LanguageRootFolder: LanguageRootFolders␣
↪should be migrate to DexterityContainers
  from plone.app.multilingual.content.lrf import LanguageRootFolder
2015-09-24 15:51:09 INFO Plone OpenID system packages not installed, OpenID support␣
↪not available
2015-09-24 15:51:11 INFO PloneFormGen Patching plone.app.portlets␣
↪ColumnPortletManagerRenderer to not catch Retry exceptions
2015-09-24 15:51:11 INFO Zope Ready to handle requests
```

If the output says `INFO Zope Ready to handle requests` then you are in business.

If you point your browser at http://localhost:8080 you see that Plone is running. There is no Plone site yet - we will create one in chapter 6.

Now you have a working Plone site up and running and can continue with the next chapter. You can stop the running instance anytime using `ctrl + c`.

Warning: If there is an error message you should either try to fix it or use vagrant and continue in this chapter.

Installing Plone with vagrant

In order not to waste too much time with installing and debugging the differences between systems, we use a virtual machine (Ubuntu 14.04) to run Plone during the training. We rely on Vagrant and VirtualBox to give the same development environment to everyone.

Vagrant is a tool for building complete development environments. We use it together with Oracle's VirtualBox to create and manage a virtual environment.

Install VirtualBox

Vagrant uses Oracle's VirtualBox to create virtual environments. Here is a link directly to the download page: https://www.virtualbox.org/wiki/Downloads. We use VirtualBox 4.3.x

Install and configure Vagrant

Get the latest version from https://www.vagrantup.com/downloads.html for your operating system and install it.

Note: In Windows there is a bug in the recent version of Vagrant. Here are the instructions for how to work around the warning `Vagrant could not detect VirtualBox! Make sure VirtualBox is properly installed`.

Now your system has a command **`vagrant`** that you can run in the terminal.

Note: You don't need to install `NodeJS` as mentioned in the previous section. Our Vagrant configuration already does that for you.

First, create a directory in which you want to do the training.

Warning: If you already have a `training` directory because you followed the **Installing Plone without vagrant** instructions above, you should either delete it, rename it, or use a different name below.

```
$ mkdir training
$ cd training
```

Setup Vagrant to automatically install the current guest additions. You can choose to skip this step if you encounter any problems with it.

```
$ vagrant plugin install vagrant-vbguest
```

Now download `plone_training_config.zip` and copy its contents into your training directory.

```
$ wget https://raw.githubusercontent.com/plone/training/master/_static/plone_training_
↪config.zip
$ unzip plone_training_config.zip
```

The training directory should now hold the file `Vagrantfile` and the directory `manifests` which again contains several files.

Now start setting up the VM that is configured in `Vagrantfile`:

```
$ vagrant up
```

This takes a **veeeeery loooong time** (between 10 minutes and 1h depending on your internet connection and system speed) since it does all the following steps:

- downloads a virtual machine (Official Ubuntu Server 14.04 LTS, also called "Trusty Tahr")
- sets up the VM
- updates the VM
- installs various system-packages needed for Plone development
- downloads and unpacks the buildout-cache to get all the eggs for Plone
- clones the training buildout into /vagrant/buildout
- builds Plone using the eggs from the buildout-cache

Note: Sometimes this stops with the message:

```
Skipping because of failed dependencies
```

If this happens or you have the feeling that something has gone wrong and the installation has not finished correctly for some reason you need to run the following command to repeat the process. This will only repeat steps that have not finished correctly.

```
$ vagrant provision
```

You can do this multiple times to fix problems, e.g. if your network connection was down and steps could not finish because of this.

Note: If while bringing vagrant up you get an error similar to:

```
ssh_exchange_identification: read: Connection reset by peer
```

The configuration may have stalled out because your computer's BIOS requires virtualization to be enabled. Check with your computer's manufacturer on how to properly enable virtualization. See: https://teamtreehouse.com/community/vagrant-ssh-sshexchangeidentification-read-connection-reset-by-peer

Once Vagrant finishes the provisioning process, you can login to the now running virtual machine.

```
$ vagrant ssh
```

Note: If you use Windows you'll have to login with putty. Connect to vagrant@127.0.01 at port 2222. User **and** password are `vagrant`.

You are now logged in as the user vagrant in `/home/vagrant`. We'll do all steps of the training as this user.

Instead we use our own Plone instance during the training. It is in `/vagrant/buildout/`. Start it in foreground with **`./bin/instance fg`**.

```
vagrant@training:~$ cd /vagrant/buildout
vagrant@training:/vagrant/buildout$ ./bin/instance fg
2015-09-24 15:51:02 INFO ZServer HTTP server started at Thu Sep 24 15:51:02 2015
        Hostname: 0.0.0.0
        Port: 8080
2015-09-24 15:51:05 WARNING PrintingMailHost Hold on to your hats folks, I'm a-patchin
↪'
2015-09-24 15:51:05 WARNING PrintingMailHost

******************************************************************************

Monkey patching MailHosts to print e-mails to the terminal.

This is instead of sending them.

NO MAIL WILL BE SENT FROM ZOPE AT ALL!

Turn off debug mode or remove Products.PrintingMailHost from the eggs
or remove ENABLE_PRINTING_MAILHOST from the environment variables to
return to normal e-mail sending.

See https://pypi.python.org/pypi/Products.PrintingMailHost

******************************************************************************

2015-09-24 15:51:05 INFO ZODB.blob (54391) Blob directory `.../buildout/var/
↪blobstorage` is unused and has no layout marker set. Selected `bushy` layout.
2015-09-24 15:51:05 INFO ZODB.blob (54391) Blob temporary directory '.../buildout/var/
↪blobstorage/tmp' does not exist. Created new directory.
.../.buildout/eggs/plone.app.multilingual-3.0.11-py2.7.egg/plone/app/multilingual/
↪browser/migrator.py:11: DeprecationWarning: LanguageRootFolder: LanguageRootFolders␣
↪should be migrate to DexterityContainers
```

```
  from plone.app.multilingual.content.lrf import LanguageRootFolder
2015-09-24 15:51:09 INFO Plone OpenID system packages not installed, OpenID support␣
↪not available
2015-09-24 15:51:11 INFO PloneFormGen Patching plone.app.portlets␣
↪ColumnPortletManagerRenderer to not catch Retry exceptions
2015-09-24 15:51:11 INFO Zope Ready to handle requests
```

Note: In rare cases when you are using OSX with an UTF-8 character set starting Plone might fail with the following error:

```
ValueError: unknown locale: UTF-8
```

In that case you have to put the localized keyboard and language settings in the .bash_profile of the vagrant user to your locale (like `en_US.UTF-8` or `de_DE.UTF-8`)

```
export LC_ALL=en_US.UTF-8
export LANG=en_US.UTF-8
```

Now the Zope instance we're using is running. You can stop the running instance anytime using `ctrl + c`.

If it doesn't, don't worry, your shell isn't blocked. Type `reset` (even if you can't see the prompt) and press RETURN, and it should become visible again.

If you point your local browser at http://localhost:8080 you see that Plone is running in vagrant. This works because VirtualBox forwards the port 8080 from the guest system (the vagrant Ubuntu) to the host system (your normal operating system). There is no Plone site yet - we will create one in chapter 6.

The Buildout for this Plone is in a shared folder. This means we run it in the vagrant box from `/vagrant/buildout` but we can also access it in our own operating system and use our favorite editor. You will find the directory `buildout` in the directory `training` that you created in the very beginning next to `Vagrantfile` and `manifests`.

Note: The database and the python packages are not accessible in your own system since large files cannot make use of symlinks in shared folders. The database lies in `/home/vagrant/var`, the python packages are in `/home/vagrant/packages`.

If you have any problems or questions please mail us at team@starzel.de or create a ticket at https://github.com/plone/training/issues.

What Vagrant does

Installation is done automatically by vagrant and puppet. If you want to know which steps are actually done please see the chapter what_vagrant_does.

Note: Vagrant Care and Handling

Keep in mind the following recommendations for using your Vagrant virtualboxes:

- Use the **`vagrant suspend`** or **`vagrant halt`** commands to put the virtualbox to "sleep" or to "power it off" before attempting to start another Plone instance anywhere else on your machine, if it uses the same port. That's because vagrant "reserves" port 8080, and even if you stopped Plone in vagrant, that port is still in use by the guest OS.

- If you are done with a vagrant box, and want to delete it, always remember to run **vagrant destroy** on it before actually deleting the directory containing it. Otherwise you'll leave its "ghost" in the list of boxes managed by vagrant and possibly taking up disk space on your machine.
- See **vagrant help** for all available commands, including **suspend**, **halt**, **destroy**, **up**, **ssh** and **resume**.

The Case Study

For this training we will build a website for a fictional Plone conference.

Background

The Plone conference takes place every year and all Plone developers at least try to go there.

Requirements

Here are some requirements that we want to meet when the site is done:

- As a visitor I want to find current information on the conference.
- As a visitor I want to register for the conference.
- As a visitor I want to see the talks and sort them by my preferences.
- As a speaker I want to be able to submit talks.
- As a speaker I want to see and edit my submitted talks.
- As an organizer I want to see a list of all proposed talks.
- As an organizer I want to have an overview about how many people registered.
- As a jury member I want to vote on talks.
- As a jury member I want to decide which talks to accept, and which not.

Note that all of our requirements connect roles with capabilities. This is important because we'll want to limit the capabilities to those to whom we assign particular roles.

The Features of Plone

In-depth user-manual: http://docs.plone.org

See also: http://docs.plone.org/working-with-content/index.html

Starting and Stopping Plone

We control Plone with a small script called "instance":

```
$ ./bin/instance fg
```

This starts Plone in foreground mode so that we can see what it is doing by monitoring console messages. This is an important development method. Note that when Plone is started in foreground mode, it is also automatically in development mode. Development mode gives better feedback, but is much slower, particularly on Windows.

You can stop it by pressing `ctrl + c`.

Apart from the *fg* command the **`instance`** script offers several more commands. *./bin/instance help* shows the list of available commands, *bin/instance help <command>* will give a short help for each command. Some commands you will use rather often are:

```
$ ./bin/instance fg
$ ./bin/instance start
$ ./bin/instance stop
$ ./bin/instance debug
$ ./bin/instance run myscript.py
$ ./bin/instance adduser name password
```

Depending on your computer, it might take up to a minute until Zope will tell you that it's ready to serve requests. On a decent laptop it should be running in under 15 seconds.

A standard installation listens on port 8080, so lets have a look at our Zope site by visiting http://localhost:8080

As you can see, there is no Plone site yet!

We have a running Zope with a database but no content. But luckily there is a button to create a Plone site. Click on that button (login: admin, password: admin). This opens a form to create a Plone site. Use `Plone` as the site id.

You now have the option to select some add-ons before you create the site. Since we will use Dexterity from the beginning we select *Dexterity-based Plone Default Types*. This way even the initial content on our page will be built with Dexterity using the add-on `plone.app.contenttypes` which is the default in Plone 5.

You will be automatically redirected to the new site.

Note: Plone has many message boxes. They contain important information. Read them and make sure you understand them!

Exercises

Exercise 1

Open the *bin/instance* script in your favorite editor. Now let's say you want Plone to listen on port 9080 instead of the default 8080. Looking at the script, how could you do this?

Solution

At the end of the *bin/instance* script, you'll see the following code:

```
if __name__ == '__main__':
    sys.exit(plone.recipe.zope2instance.ctl.main(
        ['-C', '/home/vagrant/training/buildout/parts/instance/etc/zope.conf']
        + sys.argv[1:]))
```

The second to last line points to the configuration file your Plone instance is using. An absolute path is used so it might differ depending on the installation method. Open the *zope.conf* file in your editor and look for the section:

```
<http-server>
 address 8080
</http-server>
```

Change the address to 9080 and restart your instance.

Exercise 2

Knowing that *bin/instance debug* basically offers you a Python prompt, how would you start to explore Plone?

Solution

Use *locals()* or *locals().keys()* to see Python objects available in Plone

Exercise 3

The *app* object you encountered in the previous exercise can be seen as the root of Plone. Once again using Python, can you find your newly created Plone site?

Solution

app.__dict__.keys() will show *app*'s attribute names - there is one called *Plone*, this is your Plone site object. Use *app.Plone* to access and further explore it.

Note: Plone and its objects are stored in an object database, the ZODB. You can use *bin/instance debug* as a database client (in the same way e.g. *psql* is a client for PostgreSQL). Instead of a special query language (like SQL) you simply use Python to access and manipulate ZODB objects. Don't worry if you accidentally change objects in *bin/instance debug* - you would have to commit your changes explicitly to make them permanent. The Python code to do so is:

```
>>> import transaction
>>> transaction.commit()
```

You have been warned.

Walkthrough of the UI

Let's see what is there...

- *header*:
  - *logo*: with a link to the front page
  - *searchbox*: search (with live-search)
- *navigation*: The global navigation
- *banner*: A banner. Only visible on the front page.
- *portal-columns*: a container holding:

  - *portal-column-one*: portlets (configurable boxes with tools like navigation, news etc.)
  - *portal-column-content*: the content and the editor
  - *portal-column-two*: portlets
- *portal-footer*: portlets for the footer, site actions, and colophon
- *edit-zone*: a vertical bar on the left side of the browser window with editing options for the content

These are also the CSS classes of the respective divs. If you want to do theming, you'll need them.

On the edit bar, we find options affecting the current context...

- *folder contents*
- *edit*
- *view*
- *add*
- *state*
- *actions*
- *display*
- *manage portlets*
- *history*
- *sharing*
- *rules*
- *user actions*

Some edit bar options only show when appropriate; for example, *folder contents* and *add* are only shown for Folders. *rules* is currently invisible because we have no content rules available.

Users

Let's create our first users within Plone. So far we used the admin user (admin:admin) configured in the buildout. This user is often called "Zope root" and is not managed in Plone but only by Zope. Therefore the user is missing some features like email and full name and won't be able to use some of Plone's features. But the user has all possible permissions. As with the root user of a server, it's bad practice to make unnecessary use of Zope root. Use it to create Plone sites and their initial users, but not much else.

You can also add Zope users via the terminal by entering:

```
$ ./bin/instance adduser <someusername> <supersecretpassword>
```

That way you can access databases you get from customers where you have no Plone user.

To add a new user in Plone, click on the user icon at the bottom of the left vertical bar and then on *Site setup*. This is Plone's control panel. You can also access it by browsing to http://localhost:8080/Plone/@@overview-controlpanel

Click on *Users and Groups* and add a user. If we had configured a mail server, Plone could send you a mail with a link to a form where you can choose a password. (Or, if you have Products.PrintingMailHost in your buildout, you can see the email scrolling by in the console, just the way it would be sent out.) We set a password here because we haven't yet configured a mail server.

Make this user with your name an administrator.

Then create another user called `testuser`. Make this one a normal user. You can use this user to see how Plone looks and behaves to users that have no admin permissions.

Now let's see the site in 3 different browsers with three different roles:

- as anonymous
- as editor
- as admin

Configure a Mailserver

We have to configure a mailserver since later we will create some content rules that send emails when new content is put on our site.

- Server: `localhost`
- Username: leave blank
- Password: leave blank
- Site 'From' name: Your name
- Site 'From' address: Your email address

Click on *Save and send test e-mail*. Since we have configured PrintingMailHost, you will see the mail content in the console output of your instance. Plone will not actually send the email to the receivers address.

Content-Types

Edit a page:

- *Edit front-page*
- *Title* `Plone Conference 2015,Bucharest`
- *Summary* `Tutorial`
- *Text* `...`

Create a site structure:

- Add a folder "The Event" and in it add:
  - Folder "Talks"
  - Folder "Training"
  - Folder "Sprint"
- In `/news`: Add a News Item "Conference Website online!" with some image
- In `/news`: Add a News Item "Submit your talks!"
- In `/events`: Add an Event "Deadline for talk submission" Date: 2015/08/10
- Add a Folder "Register"
- Delete the Folder "Users"
- Add a Folder "Intranet"

The default Plone content types are:

- Collection

- Event
- File
- Folder
- Image
- Link
- News Item
- Page

Note: Please keep in mind that we use plone.app.contenttypes for the training, which are the default in Plone 5. Therefore the types are based on Dexterity and slightly different from the types that you will find in a default Plone 4.3.x site.

Folders

- Go to 'the-event'
- explain the difference between title, ID, and URL
- explain /folder_contents
- change the order of items
- explain bulk actions
- dropdown "display"
- default pages
- Add a page to 'the-event': "The Event" and make it the default page

Collections

- add a new collection: "all content that has `pending` as wf_state".
- explain the default collection for events at http://localhost:8080/Plone/events/aggregator/edit
- explain Topics
- mention collection portlets
- multi-path queries
- constraints, e.g. `/Plone/folder::1`

Content Rules

- Create new rule "a new talk is in town"!
- New content in folder "Talks" -> Send Mail to reviewers.

History

Show and explain; mention versioning and its relation to types.

Manage members and groups

- add/edit/delete Users
- roles
- groups
  - Add group "Editors" and add the user 'editor' to it
  - Add group: `orga`
  - Add group: `jury` and add user 'jurymember' to it.

Workflows

Take a look at the *state* drop down on the edit bar on the homepage. Now, navigate to one of the folders just added. The homepage has the status `published` and the new content is `private`.

Let's look at the state transitions available for each type. We can make a published item private and a private item published. We can also submit an item for review.

Each of these states connects roles to permissions.

- In `published` state, the content is available to anonymous visitors;
- In `private` state, the content is only viewable by the author (owner) and users who have the `can view` role for the content.

A *workflow state* is an association between a role and one or more permissions. Moving from one state to another is a `transition`. Transitions (like `submit for review`) may have actions — such as the execution of a content rule or script — associated with them.

A complete set of workflow states and transitions makes up a *workflow*. Plone allows you to select among several pre-configured workflows that are appropriate for different types of sites. Individual content types may have their own workflow. Or, and this is particularly interesting, they may have no workflow. In that case, which initially applies to file and image uploads, the content object inherits the workflow state of its container.

Note: An oddity in all of the standard Plone workflows: a content item may be viewable even if its container is not. Making a container private does **not** automatically make its contents private.

Read more at: http://docs.plone.org/working-with-content/collaboration-and-workflow/index.html

Working copy

Published content, even in an intranet setting, can pose a special problem for editing. It may need to be reviewed before changes are made available. In fact, the original author may not even have permission to change the document without review. Or, you may need to make a partial edit. In either case, it may be undesirable for changes to be immediately visible.

Plone's working copy support solves this problem by adding a check-out/check-in function for content — available on the actions menu. A content item may be checked out, worked on, then checked back in. Or it may abandoned if the changes weren't acceptable. Not until check in is the new content visible.

While it's shipped with Plone, working copy support is not a common need. So, if you need it, you need to activate it via the add-on packages configuration page. Unless activated, check-in/check-out options are not visible.

Note: Working-copy support is not yet available for content types created via Dexterity. This is on the way.

Placeful workflows

You may need to have different workflows in different parts of a site. For example, we created an intranet folder. Since this is intended for use by our conference organizers — but not the public — the simple workflow we wish to use for the rest of the site will not be desirable.

Plone's `Workflow Policy Support` package gives you the ability to set different workflows in different sections of a site. Typically, you use it to set a special workflow in a folder that will govern everything under that folder. Since it has effect in a "place" in a site, this mechanism is often called "Placeful Workflow".

As with working-copy support, Placeful Workflow ships with Plone but needs to be activated via the add-on configuration page. Once it's added, a *Policy* option will appear on the state menu to allow setting a placeful workflow policy.

Note: Workflow Policy support is not yet available for folderish contenttypes created via Dexterity. This is on the way.

The Anatomy of Plone

In this part you will:

- Learn a bit about the history of Plone.

Topics covered:

- CMF
- Zope
- Pyramid
- Bluebream

Python, Zope, CMF, Plone ... – how does all that fit together?

Zope2

- Zope is a web application framework that Plone runs on top of.
- The majority of Zope's code is written in Python, like everything else written on top of it.
- It serves applications that communicate with users via http.

Before Zope, there usually was an Apache server that would call a script and give the request as an input. The script would then just print HTML to the standard output. Apache returned that to the user. Opening database connections, checking permission constraints, generating valid HTML, configuring caching, interpreting form data and everything else: you have to do it on your own. When the second request comes in, you have to do everything again.

Jim Fulton thought that this was slightly tedious. So he wrote code to handle requests. He believed that site content is object-oriented and that the URL should somehow point directly into the object hierarchy, so he wrote an object-oriented database, called ZODB.

The ZODB is a fully ACID compliant database with automatic transactional integrity. It automatically maps traversal in the object hierarchy to URL paths, so there is no need to "wire" objects or database nodes to URLs. This gives Plone its easy SEO-friendly URLs.

Traversal through the object database is security checked at every point via very fine grained access-control lists.

One missing piece is important and complicated: `Acquisition`.

Acquisition is a kind of magic. Imagine a programming system where you do not access the file system and where you do not need to import code. You work with objects. An object can be a folder that contains more objects, an HTML page, data, or another script. To access an object, you need to know where the object is. Objects are found by paths that look like URLs, but without the domain name. Now Acquisition allows you to write an incomplete path. An incomplete path is a relative path, it does not explicitly state that the path starts from the root, it starts relative to where the content object is – its context. If Zope cannot resolve the path to an object relative to your code, it tries the same path in the containing folder. And then the folder containing the folder.

This might sound weird, what do I gain with this?

You can have different data or code depending on your `context`. Imagine you want to have header images differing for each section of your page, sometimes even differing for a specific subsection of your site. So you define a path `header_image` and put a header image at the root of your site. If you want a folder with a different header image, you put the header image into this folder. Please take a minute to let this settle and think about what this allows you to do.

- contact forms with different e-mail addresses per section
- different CSS styles for different parts of your site
- One site, multiple customers, everything looks different for each customer.

As with all programming magic, acquisition exacts a price. Zope code must be written carefully in order to avoid inheriting side effects via acquisition. The Zope community expresses this with the Python (Monty) maxim: Beware the *Spammish Acquisition*.

Basically this is Zope.

See also:

- http://www.zope.org/en/latest/world.html
- http://docs.zope.org/zope2/zope2book/

Content Management Framework

- CMF (Content Management Framework) is add-on for Zope to build Content Management Systems (like Plone).

After many websites were successfully created using Zope, a number of recurring requirements emerged, and some Zope developers started to write CMF, the Content Management Framework.

The CMF offers many services that help you to write a CMS based on Zope. Most objects you see in the ZMI are part of the CMF somehow.

The developers behind CMF do not see CMF as a ready to use CMS. They created a CMS Site which was usable out of the box, but made it deliberately ugly, because you have to customize it anyway.

We are still in prehistoric times here. There were no eggs (Python packages), Zope did not consist of 100 independent software components but was one big file set.

Many parts of Plone are derived from the CMF, but it's a mixed heritage. The CMF is an independent software project, and has often moved more slowly than Plone. Plone is gradually eliminating dependence on most parts of the CMF.

Zope Toolkit / Zope3

- Zope 3 was originally intended as a rewrite of Zope from the ground up.
- Plone uses parts of it provided by the Zope Toolkit (ZTK).

Unfortunately, only few people started to use Zope 3, nobody migrated to Zope 3 because nobody knew how.

But there were many useful things in Zope 3 that people wanted to use in Zope 2, thus the Zope community adapted some parts so that they could use them in Zope 2. Sometimes, a wrapper of some sort was necessary, these usually are being provided by packages from the `five` namespace. (Zope 2 + Zope 3 = "five")

To make the history complete, since people stayed on Zope 2, the Zope community renamed Zope 3 to Bluebream, so that people would not think that Zope 3 was the future. It wasn't anymore.

Zope Component Architecture (ZCA)

The Zope Component Architecture, which was developed as part of Zope 3, is a system which allows for component pluggability and complex dispatching based on objects which implement an interface (a description of a functionality). It is a subset of the ZTK but can be used standalone. Plone makes extensive use of the ZCA in its codebase.

Pyramid

- Pyramid is a Python web application development framework that is often seen as the successor to Zope.
- It does less than Zope, is very pluggable and uses the Zope Component Architecture "under the hood" to perform view dispatching and other application configuration tasks.

You can use it with a relational Database instead of ZODB if you want, or you can use both databases or none of them.

Apart from the fact that Pyramid was not forced to support all legacy functionality, which can make things more complicated, the original developer had a very different stance on how software must be developed. While both Zope and Pyramid have good test coverage, Pyramid has good documentation; something that was very neglected in Zope, and at times in Plone too.

Whether the component architecture is better in Pyramid or not we don't dare say, but we like it more. But maybe it's just because it was documented.

See also:

- http://docs.pylonsproject.org/projects/pyramid/en/latest/index.html

Exercise

Definition of the PYTHON_PATH makes up most of the *bin/instance* script's code. Look at the package list (and maybe also the links provided in the respective sections of this chapter). Try to identify 3 packages that belong to the original Zope2, 3 packages from CMF, 3 Zope Toolkit packages and 3 packages from the ZCA.

Solution

- Zope2: Zope2, ZODB, Acquistion, AccessControl, ...
- CMF: Products.CMFCore, Products.CMFUid, Products.CMFEditions, ... Products.DCWorkflow doesn't fit the pattern but is a very important part of the CMF
- ZTK: zope.browser, zope.container, zope.pagetemplate, ... You can find a complete list herehttps://dist.plone.org/versions/zopetoolkit-1-0-8-zopeapp-versions.cfg

- ZCA: zope.component, zope.interface, zope.event

What's New in Plone 5

If you are already used to Plone 5 you could skip this section.

Default Theme

The new default theme is called Barceloneta

It is a Diazo theme, meaning it uses `plone.app.theming` to insert the output of Plone into static html/css.

It uses html5, so it uses `<header>`, `<nav>`, `<aside>`, `<section>`, `<article>` and `<footer>` for semantic html.

The theme is mostly built with LESS (lots of it!) and uses the same grid system as bootstrap. This means you can use css classes like `col-xs-12 col-sm-9` to control the width of elements for different screen-sizes. If you prefer a different grid-system (like foundation) over bootstrap you can adapt the theme to use that.

The index.html and rules.xml are actually not that complicated. Have a look at them.

The following example from `rules.xml` makes sure that the banner saying *"Welcome! Plone 5 rocks!"* is only visible on the frontpage:

```
<!-- 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" />
```

The browser-view `@@hero` (you can find it by searching all ZCML-files for `name="hero"`) is only included when the body-tag of the current page has the css-classes `template-document_view` and `section-front-page`.

New UI and widgets

The green edit bar is replaced by a toolbar that is located on the left or top and can be expanded. The design of the toolbar is pretty isolated from the theme and it should not break if you use a different theme.

The widgets where you input data are also completely rewritten.

- We now use the newest TinyMCE
- The tags (keywords) widget and the widgets where you input usernames now use select2 autocomplete to give a better user experience
- The related-items widget is a complete rewrite

Folder Contents

The view to display the content of a folder is new and offers many new features:

- configurable table columns
- changing properties of multiple items at once
- querying (useful for folders with a lot of content)

- persistent selection of items

Content Types

All default types are based on Dexterity. This means you can use behaviors to change their features and edit them through the web. Existing old content can be migrated to these types.

Resource Registry

The resource registry allows you to configure and edit the static resources (js, css) of Plone. It replaces the old javascript and css registries. And it can be used to customize the theme by changing the variables used by LESS or overriding LESS files.

Chameleon template engine

Chameleon is the new rendering engine of Plone 5. It offers many improvements:

Old syntax:

```
<h1 tal:attributes="title view/title"
    tal:content="view/page_name">
</h1>
```

New (additional) syntax:

```
<h1 title="${view/title}">
    ${view/page_name}
</h1>
```

Template debugging:

You can now put a full-grown `pdb` in a template.

```
<?python import pdb; pdb.set_trace() ?>
```

For debugging check out the variable `econtext`, it holds all the current elements.

You can also add real Python blocks inside templates.

```
<?python

from plone import api

catalog = api.portal.get_tool('portal_catalog')
results = []
for brain in catalog(portal_type='Folder'):
    results.append(brain.getURL())

?>

<ul>
    <li tal:repeat="result results">
      ${result}
    </li>
</ul>
```

Don’t overdo it!

Control panel

- You can finally upload a logo in `@@site-controlpanel`.
- All control panels were moved to z3c.form
- Many small improvements

Date formatting on the client side

Using the js library moment.js the formatting of dates was moved to the client.

```
<ul class="pat-moment"
    data-pat-moment="selector:li;format:calendar;">
    <li>${python:context.created().ISO()}</li>
    <li>2015-10-22T12:10:00-05:00</li>
</ul>
```

returns

- Today at 3:24 PM
- 10/22/2015

plone.app.multilingual

plone.app.multilingual is the new default add-on for sites in more than one language.

New portlet manager

`plone.footerportlets` is a new place to put portlets. The footer (holding the footer, site_actions, colophon) is now built from portlets. This means you can edit the footer TTW.

There is also a useful new portlet type *Actions* used for displaying the site_actions.

Remove portal_skins

Many of the old skin templates were replaced by real browser views.

Configuring and Customizing Plone “Through The Web”

Warning:

> This chapter has not yet been updated for Plone 5!

The Control Panel

The most important parts of Plone can be configured in the control panel.

- Click on the portrait/username in the toolbar
- Click *Site Setup*

We'll explain every page and mention some of the actions you can perform here.

General

1. Date and Time
2. Language
3. Mail
4. Navigation
5. Site
6. Add-ons
7. Search
8. Discussion
9. Theming
10. Social Media
11. Syndication
12. TinyMCE

Content

1. Content Rules
2. Editing
3. Image Handling
4. Markup
5. Content Settings
6. Dexterity Content Types

Users

1. Users and Groups

Security

1. HTML Filtering
2. Security
3. Errors

Advanced

1. Maintenance
2. Management Interface
3. Caching
4. Configuration Registry
5. Resource Registries

Below the links you will find information on your Plone, Zope and Python Versions and an indicator as to whether you're running in production or development mode.

Change the logo

Let's change the logo.

- Download a ploneconf logo: https://www.starzel.de/plone-tutorial/ploneconf-logo-2014
- Go to http://localhost:8080/Plone/@@site-controlpanel
- Upload the Logo.

See also:

http://docs.plone.org/adapt-and-extend/change-the-logo.html

Portlets

In the toolbar under *More options* you can open the configuration for the different places where you can have portlets.

- UI fit for smart content editors
- Various types
- Portlet configuration is inherited
- Managing
- Ordering/weighting
- The future: may be replaced by tiles
- `@@manage-portlets`

Example:

- Go to http://localhost:8080/Plone/@@manage-portlets
- Add a static portlet "Sponsors" on the right side.
- Remove the news portlet and add a new one on the left side.
- Go to the training folder: http://localhost:8080/Plone/the-event/training and click `Manage portlets`
- Add a static portlet. "Featured training: Become a Plone-Rockstar at Mastering Plone!"
- Use the toolbar to configure the portlets of the footer:
  - Hide the portlets "Footer" and "Colophon".
  - Add a "Static text portlet" enter "Copyright 2015 by Plone Community".

  - Use “Insert > Special Character” to add a real © sign.
  - You could turn that into a link to a copyright page later.

Viewlets

Portlets save data, Viewlets usually don’t. Viewlets are often used for UI-Elements and have no nice UI to customize them.

- `@@manage-viewlets`
- Viewlets have no nice UI
- Not aimed at content editors
- Not locally addable, no configurable inheritance.
- Usually global (depends on code)
- Will be replaced by tiles?
- The code is much simpler (we’ll create one tomorrow).
- Live in viewlet managers, can be nested (by adding a viewlet that contains a viewlet manager).
- TTW reordering only within the same viewlet manager.
- The code decides when it is shown and what it shows.

ZMI (Zope Management Interface)

Go to http://localhost:8080/Plone/manage

Zope is the foundation of Plone. Here you can access the inner workings of Zope and Plone alike.

Note: Here you can easily break your site so you should know what you are doing!

We only cover three parts of customization in the ZMI now. Later on when we added our own code we’ll come back to the ZMI and will look for it.

At some point you’ll have to learn what all those objects are about. But not today.

Actions (portal_actions)

- Actions are mostly links. But **really flexible** links.
- Actions are configurable ttw and through code.
- These actions are usually iterated over in viewlets and displayed.

Examples:

- Links in the Footer (`site_actions`)
- Actions Dropdown (`folder_buttons`)

Actions have properties like:

- description
- url

- i18n-domain
- condition
- permissions

`site_actions`

These are the links at the bottom of the page:

- Site Map
- Accessibility
- Contact
- Site Setup

We want a new link to legal information, called "Imprint".

- Go to `site_actions` (we know that because we checked in `@@manage-viewlets`)
- Add a CMF Action `imprint`
- Set URL to `string:${portal_url}/imprint`
- Leave *condition* empty
- Set permission to `View`
- Save

explain

- Check if the link is on the page
- Create new Document *Imprint* and publish

See also:

http://docs.plone.org/develop/plone/functionality/actions.html

Global navigation

- The horizontal navigation is called `portal_tabs`
- Go to *portal_actions* → *portal_tabs* Link
- Edit `index_html`

Where is the navigation?

The navigation shows content-objects, which are in Plone's root. Plus all actions in `portal_tabs`.

Explain & edit `index_html`

Configuring the navigation itself is done elsewhere: http://localhost:8080/Plone/@@navigation-controlpanel

If time explain:

- user > undo (cool!)
- user > login/logout

Skins (portal_skins)

In portal_skins we can change certain images, CSS-files and templates.

- portal_skins is deprecated technology
- Plone 5 got rid of most files that lived in portal_skins.

Change some CSS

- Go to ZMI
- Go to portal_skins
- Go to plone_styles
- Go to ploneCustom.css
- Click *customize*

The CSS you add to this file is instantly active on the site.

portal_view_customizations

Change the footer

- Go to portal_view_customizations
- Search plone.footer, click and customize
- Replace the content with the following

```
<div i18n:domain="plone"
     id="portal-footer">
   <p>&copy; 2016 by me! |
     <a href="mailto:info@ploneconf.org">
      Contact us
     </a>
   </p>
</div>
```

See also:

http://docs.plone.org/adapt-and-extend/theming/templates_css/skin_layers.html

CSS Registry (portal_css)

deprecated (See the chapter on theming)

Further tools in the ZMI

There are many more notable items in the ZMI. We'll visit some of them later.

- *acl_users*
- *error_log*

- *portal_properties* (deprecated)
- *portal_setup*
- *portal_workflow*
- *portal_catalog*

Summary

You can configure and customize a lot in Plone through the web. The most important options are accessible in the Plone control panel but some are hidden away in the ZMI. The amount and presentation of information is overwhelming but you'll get the hang of it through a lot of practice.

Theming

We don't do any real theming during the training. We'll just explain the options you have.

If you really want to learn about theming see http://docs.plone.org/adapt-and-extend/theming/index.html and the Training *Mastering Plone Theming*

Extending Plone

In this part you will:

- Get an overview over the technologies used to extend Plone

Topics covered:

- Skin folders
- GenericSetup
- Component Architecture
- ZCML

Zope is extensible and so is Plone.

If you want to install an add-on, you are going to install an Egg — a form of Python package. Eggs consist of Python files together with other needed files like page templates and the like and a bit of metadata, bundled to a single archive file.

There is a huge variety of Plone-compatible packages available. See Plone.org add-on listing. The source repository for many public Plone add-ons is the GitHub Collective. You may also create your own packages or maintain custom repositories.

Eggs are younger than Zope. Zope needed something like eggs before there were eggs, and the Zope developers wrote their own system. Old, outdated Plone systems contain a lot of code that is not bundled in an egg. Older code did not have metadata to register things, instead you needed a special setup method. We don't need this method but you might see it in other code. It is usually used to register Archetypes code. Archetypes is the old content type system. Instead, we use the new content type system Dexterity.

Extension technologies

How do you extend Plone?

This depends on what type of extension you want to create.

- You can create extensions with new types of objects to add to your Plone site. Usually these are contenttypes.
- You can create an extension that changes or extends functionality. For example to change the way Plone displays search results, or to make pictures searchable by adding a converter from jpg to text.

Skin Folders

Do you remember Acquisition? The Skin Folders extends the concepts of Acquisition. Your Plone site has a folder named `portal_skins`. This folder has a number of sub folders. The `portal_skins` folder has a property that defines in which order Plone searches for attributes or objects in each sub folder.

The Plone logo is in a skin folder.

By default, your site has a `custom` folder, and items are first searched for in that folder.

To customize the logo, you copy it into the `custom` folder, and change it there. This way you can change templates, CSS styles, images and behavior, because a container may contain Python scripts.

Skin-folder style customization may be accomplished TTW via the ZMI, or with add-on packages. Many older-style packages create their own skin folder and add it to the skin layer for Plone when installed.

Warning: This is deprecated technology.

GenericSetup

The next thing is *GenericSetup*. As the name clearly implies, *GenericSetup* is part of CMF.

GenericSetup is tough to master, I am afraid.

GenericSetup lets you define persistent configuration in XML files. *GenericSetup* parses the XML files and updates the persistent configuration according to the configuration. This is a step you have to run on your own!

You will see many objects in Zope or the ZMI that you can customize through the web. If they are well behaving, they can export their configuration via *GenericSetup* and import it again.

Typically you use *GenericSetup* to change workflows or add new content type definitions.

GenericSetup profiles may also be built into Python packages. Every package that is listed on the add-on package list inside a Plone installation has a GS profile that details how it fits into Plone. Packages that are part of Plone itself may have GS profiles, but are excluded from the active/inactive listing.

Component Architecture

The last way to extend Plone is via *Components*.

A bit of history is in order.

When Zope started, object-oriented design was **the** silver bullet.

Object-oriented design is good at modeling inheritance, but breaks down when an object has multiple aspects that are part of multiple taxonomies.

Some object-oriented programming languages like Python handle this through multiple inheritance. But it's not a good way to do it. Zope objects have more than 10 base classes. Too many namespaces makes code that's hard to maintain. Where did that method/attribute come from?

After a while, XML and Components became the next silver bullet (Does anybody remember J2EE?).

Based on their experiences with Zope in the past, Zope developers thought that a component system configured via XML might be the way to go to keep the code more maintainable.

As the new concepts were radically different from the old Zope concepts, the Zope developers renamed the new project to Zope 3. But it did not gain traction, the community somehow renamed it to Bluebream and this died off.

But the component architecture itself is quite successful and the Zope developers extracted it into the Zope Toolkit. The Zope toolkit is part of Zope, and Plone developers use it extensively.

This is what you want to use.

What are components, what is ZCML

What is the absolute simplest way to extend functionality?

Monkey Patching.

It means that you change code in other files while my file gets loaded.

If you want to have an extensible registry of icons for different contenttypes, you could create a global dictionary, and whoever implements a new icon for a different content type would add an entry to my dictionary during import time.

This approach, like subclassing via multiple inheritance, does not scale. Multiple plugins might overwrite each other, you would explain to people that they have to reorder the imports, and then, suddenly, you will be forced to import feature A before B, B before C and C before A, or else your application won't work.

The Zope Component Architecture with its ZCML configuration is an answer to these problems.

With ZCML you declare utilities, adapters and browser views in ZCML, which is an XML dialect. ZCML stands for Zope Component Markup Language.

Components are differentiated from one another by the interfaces (formal definitions of functionality) that they require or provide.

During startup, Zope reads all these ZCML statements, validates that there are not two declarations trying to register the same components and only then registers everything. All components are registered by interfaces required and provided. Components with the same interfaces may optionally also be named.

This is a good thing. ZCML is, by the way, only *one* way to declare your configuration.

Grok provides another way, where some Python magic allows you to use decorators to register Python classes and functions as components. You can use ZCML and Grok together if you wish.

Some like Grok because it allows you to do nearly everything in your Python source files. No additional XML wiring required. If you're XML-allergic, Grok is your ticket to Python nirvana.

Not everybody loves Grok. Some parts of the Plone community think that there should only be one configuration language, others are against adding the relative big dependency of Grok to Plone. One real problem is the fact that you cannot customize components declared with grok with jbot (which we'll discuss later). Grok is not allowed in the Plone core for these reasons.

The choice to Grok or not to Grok is yours to make. In any case, if you start to write an extension that is reusable, convert your grok declarations to ZCML to get maximum acceptance.

Personally, I just find it cumbersome but even for me as a developer it offers a nice advantage: thanks to ZCML, I hardly ever have a hard time to find what and where extensions or customizations are defined. For me, ZCML files are like a phone book.

Extend Plone With Add-On Packages

- There are more than 2,000 add-ons for Plone. We will cover only a handful today.

- Using them saves a lot of time
- The success of a project often depends on finding the right add-on
- Their use, usefulness, quality and complexity varies a lot

Some notable add-ons

Products.PloneFormGen A form generator.

collective.disqus Integrates the Disqus commenting platform API into Plone

collective.plonetruegallery Photo galleries with a huge selection of various js-libraries.

plone.app.mosaic Layout solution to easily create complex layouts through the web.

collective.geo Flexible bundle of add-ons to geo-reference content and display in maps

collective.mailchimp Allows visitors to subscribe to mailchimp newsletters

eea.facetednavigation Create faceted navigation and searches through the web.

collective.lineage Microsites for Plone - makes subfolders appear to be autonomous Plone sites

Products.Doormat A flexible doormat

collective.behavior.banner Add decorative banners and sliders

plone.app.multilingual Allows multilingual sites by translating content.

Rapido Allows developers with a little knowledge of HTML and a little knowledge of Python to implement custom elements and insert them anywhere they want.

Plomino Powerful and flexible web-based application builder for Plone

> **Warning:** Some add-ons may not yet run under Plone 5 and will have to be updated to be compatible.

How to find add-ons

- https://plone.org/download/add-ons
- https://pypi.python.org/pypi - use the search form!
- https://github.com/collective >1200 repos
- https://github.com/plone >260 repos
- https://community.plone.org - ask the community
- google (e.g. Plone+Slider)
- ask in irc and on the mailing list

See also:

- A talk on finding and managing add-ons: https://www.youtube.com/watch?v=Sc6NkqaSjqw

Installing Add-ons

Installation is a two-step process.

Making the add-on packages available to Zope

First, we must make the add-on packages available to Zope. This means that Zope can import the code. Buildout is responsible for this.

Look at the `buildout.cfg` file in `/vagrant/buildout`.

Note: If you're using our Vagrant kit, the Plone configuration is available in a folder that is shared between the host and guest operating systems. Look in your Vagrant install directory for the `buildout` folder. You may edit configuration files using your favorite text editor in the host operating system, then switch into your virtual machine to run buildout on the guest operating system.

In the section `[instance]` there is a variable called `eggs`, which has a list of *eggs* as a value. For example:

```
eggs =
    Plone
    Products.PloneFormGen
    plone.app.debugtoolbar
```

You add an egg by adding a new line containing the package name to the configuration. You must write the egg name indented: this way, buildout understands that the current line is part of the last variable and not a new variable.

If you add new add-ons here you will have to run buildout and restart the site:

```
$ bin/buildout
$ bin/instance fg
```

Now the code is available from within Plone.

Installing add-ons in your Plone Site

Your Plone site has not yet been told to use the add-on. For this, you have to activate the add-on in your Plone Site.

Note: Why the extra step of activating the add-on package? You may have multiple Plone sites in a single Zope installation. It's common to want to activate some add-ons in one site, others in another.

In your browser, go to Site Setup (shortcut: add `/@@overview-controlpanel` to the Plone site URL), and open the `Add-ons` Panel. You will see that you can install the add-ons there.

Install **PloneFormGen** now.

This is what happens: The GenericSetup profile of the product gets loaded. This does things like:

- Configuring new actions
- Registering new contenttypes
- Registering css and js files
- Creating some content/configuration objects in your Plone site.

Let's have a look at what we just installed.

PloneFormGen

There are many ways to create forms in Plone:

- Pure: html and python in a view
- Framework: z3c.form, formlib, deform
- TTW: Products.PloneFormGen

The basic concept of PloneFormGen is that you build a form by adding a Form Folder, to which you add form fields as content items. Fields are added, deleted, edited and moved just as with any other type of content. Form submissions may be automatically emailed and/or saved for download. There are many add-ons to PloneFormGen that provide additional field types and actions.

Let's build a registration form:

- Activate PloneFormGen for this site via the add-on configuration panel in site setup
- Add an object of the new type 'Form Folder' in the site root. Call it "Registration"
- Save and view the result, a simple contact form that we may customize
- Click in QuickEdit
- Remove field "Subject"
- Add fields for food preference and shirt size
- Add a DataSave Adapter
- Customize the mailer

Note: Need CAPTCHAs? Add the `collective.recaptcha` package to your buildout and PFG will have a CAPTCHA field.

Need encryption? Add GPG encryption to your system, add a GPG configuration for the Plone daemon user that includes a public key for the mail targets, and you'll be able to encrypt email before sending.

Think PFG is too complicated for your site editors? Administrators (and we're logged in as an administrator) see lots of more complex options that are invisible to site editors.

By the way, while PloneFormGen is good at what it does, it is not a good model for designing your own extensions. It was created before the Zope Component Architecture became widely used. The authors would write it much differently if they were starting from scratch.

Note: collective.easyform is a alternative form-generator that uses dexterity. It is still under development.

Add Photo Gallery with `collective.plonetruegallery`

To advertise the conference we want to show some photos showing past conferences and the city where the conference is taking place.

Instead of creating new contenttypes for galleries, it integrates with the Plone functionality to choose different views for folderish contenttypes.

https://pypi.python.org/pypi/collective.plonetruegallery

- Activate the add-on

- Enable the behavior `Plone True Gallery` on the type `Folder`: http://localhost:8080/Plone/dexterity-types/Folder/@@behaviors
- Add a folder `/the-event/location`
- Upload some photos from lorempixel.com
- Enable the view `galleryview`

Internationalization

Plone can run the same site in many different languages.

We're not doing this with the conference site since the *lingua franca* of the Plone community is English.

We would use the built-in addon https://pypi.python.org/pypi/plone.app.multilingual for this.

Building a multi-lingual site requires activating `plone.app.multilingual`, but no add-on is necessary to build a site in only one language. Just select a different site language when creating a Plone site, and all text in the user-interface will be switched to that language.

Summary

We are now able to customize and extend many parts of our website. We can even install extensions that add new functionality.

But:

- Can we submit talks now?
- Can we create lists with the most important properties of each talk?
- Can we allow a jury to vote on talks?

We often have to work with structured data. Up to a degree we can do all this TTW, but at some point we run into barriers. In the next part of the training, we'll teach you how to break through these barriers.

Dexterity I: "Through The Web"

In this part you will:

- Create a new content type called *Talk*.

Topics covered:

- Content types
- Archetypes and Dexterity
- Fields
- Widgets

What is a content type?

A content type is a kind of object that can store information and is editable by users. We have different content types to reflect the different kinds of information about which we need to collect and display information. Pages, folders, events, news items, files (binary) and images are all content types.

It is common in developing a web site that you'll need customized versions of common content types, or perhaps even entirely new types.

Remember the requirements for our project? We wanted to be able to solicit and edit conference talks. We *could* use the **Page** content type for that purpose. But we need to make sure we collect certain bits of information about a talk and we couldn't be sure to get that information if we just asked potential presenters to create a page. Also, we'll want to be able to display talks featuring that special information, and we'll want to be able to show collections of talks. A custom content type will be ideal.

The makings of a Plone content type

Every Plone content type has the following parts:

Schema A definition of fields that comprise a content type; properties of an object.

FTI The "Factory Type Information" configures the content type in Plone, assigns it a name, an icon, additional features and possible views to it.

Views A view is a representation of the object and the content of its fields that may be rendered in response to a request. You may have *one or more* views for an object. Some may be *visual* — intended for display as web pages — others may be intended to satisfy AJAX requests and render content in formats like JSON or XML.

Dexterity and Archetypes - A Comparison

There are two content frameworks in Plone:

- *Dexterity*: new and the coming default.
- *Archetypes*: old, tried and tested. Widespread, used in many add-ons.
- Plone 4.x: Archetypes is the default, with Dexterity available.
- Plone 5.x: Dexterity is the default, with Archetypes available.
- For both, add and edit forms are created automatically from a schema.

What are the differences?

- Dexterity: New, faster, modular, no dark magic for getters and setters.
- Archetypes had magic setter/getter - use `talk.getAudience()` for the field `audience`.
- Dexterity: fields are attributes: `talk.audience` instead of `talk.getAudience()`.

"Through The Web" or TTW, i.e. in the browser, without programming:

- Dexterity has a good TTW story.
- Archetypes has no TTW story.
- UML-modeling: ArchGenXML for Archetypes, agx for Dexterity

Approaches for Developers:

- Schema in Dexterity: TTW, XML, Python. Interface = schema, often no class needed.
- Schema in Archetypes: Schema only in Python.
- Dexterity: Easy permissions per field, easy custom forms.
- Archetypes: Permissions per field are hard, custom forms even harder.
- If you have to program for old sites you need to know Archetypes!
- If starting fresh, go with Dexterity.

Extending:

- Dexterity has Behaviors: easily extendable. Just activate a behavior TTW and your content type is e.g. translatable (`plone.app.multilingual`).
- Archetypes has `archetypes.schemaextender`. Powerful but not as flexible.

We have only used Dexterity for the last few years. We teach Dexterity and not Archetypes because it's more accessible to beginners, has a great TTW story and is the future.

Views:

- Both Dexterity and Archetypes have a default view for content types.
- Browser Views provide custom views.
- You can generate views for content types in the browser without programming (using the `plone.app.mosaic` Add-on).
- Display Forms.

Modifying existing types

- Go to the control panel http://localhost:8080/Plone/@@dexterity-types
- Inspect some of the existing default types.
- Select the type *News Item* and add a new field `Hot News` of type *Yes/No*
- In another tab, add a *News Item* and you'll see the new field.
- Go back to the schema-editor and click on Edit XML Field Model.
- Note that the only field in the XML schema of the News Item is the one we just added. All others are provided by behaviors.
- Edit the form-widget-type so it says:

```
<form:widget type="z3c.form.browser.checkbox.SingleCheckBoxFieldWidget"/>
```

- Edit the News Item again. The widget changed from a radio field to a check box.
- The new field `Hot News` is not displayed when rendering the News Item. We'll take care of this later.

See also:

http://docs.plone.org/external/plone.app.contenttypes/docs/README.html#extending-the-types

Creating content types TTW

In this step we will create a content type called *Talk* and try it out. When it's ready we will move the code from the web to the file system and into our own add-on. Later we will extend that type, add behaviors and a viewlet for Talks.

- Add new content type "Talk" and some fields for it:
  - *Add new field* "Type of talk", type "Choice". Add options: talk, keynote, training.
  - *Add new field* "Details", type "Rich Text" with a maximal length of 2000.
  - *Add new field* "Audience", type "Multiple Choice". Add options: beginner, advanced, pro.
  - Check the behaviors that are enabled: *Dublin Core metadata*, *Name from title*. Do we need them all?
- Test the content type.

- Return to the control panel http://localhost:8080/Plone/@@dexterity-types
- Extend the new type: add the following fields:
  - "Speaker", type: "Text line"
  - "Email", type: "Email"
  - "Image", type: "Image", not required
  - "Speaker Biography", type: "Rich Text"
- Test again.

Here is the complete XML schema created by our actions:

```xml
<model xmlns:lingua="http://namespaces.plone.org/supermodel/lingua"
     xmlns:users="http://namespaces.plone.org/supermodel/users"
     xmlns:security="http://namespaces.plone.org/supermodel/security"
     xmlns:marshal="http://namespaces.plone.org/supermodel/marshal"
     xmlns:form="http://namespaces.plone.org/supermodel/form"
     xmlns="http://namespaces.plone.org/supermodel/schema">
  <schema>
    <field name="type_of_talk" type="zope.schema.Choice">
      <description/>
      <title>Type of talk</title>
      <values>
        <element>Talk</element>
        <element>Training</element>
        <element>Keynote</element>
      </values>
    </field>
    <field name="details" type="plone.app.textfield.RichText">
      <description>Add a short description of the talk (max. 2000 characters)</description>
      <max_length>2000</max_length>
      <title>Details</title>
    </field>
    <field name="audience" type="zope.schema.Set">
      <description/>
      <title>Audience</title>
      <value_type type="zope.schema.Choice">
        <values>
          <element>Beginner</element>
          <element>Advanced</element>
          <element>Professionals</element>
        </values>
      </value_type>
    </field>
    <field name="speaker" type="zope.schema.TextLine">
      <description>Name (or names) of the speaker</description>
      <title>Speaker</title>
    </field>
    <field name="email" type="plone.schema.email.Email">
      <description>Adress of the speaker</description>
      <title>Email</title>
    </field>
    <field name="image" type="plone.namedfile.field.NamedBlobImage">
      <description/>
      <required>False</required>
      <title>Image</title>
    </field>
    <field name="speaker_biography" type="plone.app.textfield.RichText">
      <description/>
      <max_length>1000</max_length>
      <required>False</required>
      <title>Speaker Biography</title>
    </field>
  </schema>
</model>
```

Moving contenttypes into code

It's awesome that we can do so much through the web. But it's also a dead end if we want to reuse this content type in other sites.

Also, for professional development, we want to be able to use version control for our work, and we'll want to be able to add the kind of business logic that will require programming.

So, we'll ultimately want to move our new content type into a Python package. We're missing some skills to do that, and we'll cover those in the next couple of chapters.

See also:

- Dexterity Developer Manual
- The standard behaviors

Exercises

Exercise 1

Modify Pages to allow uploading an image as decoration (like News Items do).

Solution

- Go to the dexterity control panel (http://localhost:8080/Plone/@@dexterity-types)
- Click on *Page* (http://127.0.0.1:8080/Plone/dexterity-types/Document)
- Select the tab *Behaviors* (http://127.0.0.1:8080/Plone/dexterity-types/Document/@@behaviors)
- Check the box next to *Lead Image* and save.

The images are displayed above the title.

Exercise 2

Create a new content type called *Speaker* and export the schema to a XML File. It should contain the following fields:

- Title, type: "Text Line"
- Email, type: "Email"
- Homepage, type: "URL" (optional)
- Biography, type: "Rich Text" (optional)

- Company, type: “Text Line” (optional)
- Twitter Handle, type: “Text Line” (optional)
- IRC Handle, type: “Text Line” (optional)
- Image, type: “Image” (optional)

Do not use the DublinCore or the Basic behavior since a speaker should not have a description (unselect it in the Behaviors tab).

We could use this content type later to convert speakers into Plone users. We could then link them to their talks.

Solution

The schema should look like this:

```
<model xmlns:lingua="http://namespaces.plone.org/supermodel/lingua"
       xmlns:users="http://namespaces.plone.org/supermodel/users"
       xmlns:security="http://namespaces.plone.org/supermodel/security"
       xmlns:marshal="http://namespaces.plone.org/supermodel/marshal"
       xmlns:form="http://namespaces.plone.org/supermodel/form"
       xmlns="http://namespaces.plone.org/supermodel/schema">
  <schema>
    <field name="title" type="zope.schema.TextLine">
      <title>Name</title>
    </field>
    <field name="email" type="plone.schema.email.Email">
      <title>Email</title>
    </field>
    <field name="homepage" type="zope.schema.URI">
      <required>False</required>
      <title>Homepage</title>
    </field>
    <field name="biography" type="plone.app.textfield.RichText">
      <required>False</required>
      <title>Biography</title>
    </field>
    <field name="company" type="zope.schema.TextLine">
      <required>False</required>
      <title>Company</title>
    </field>
    <field name="twitter_handle" type="zope.schema.TextLine">
      <required>False</required>
      <title>Twitter Handle</title>
    </field>
    <field name="irc_name" type="zope.schema.TextLine">
      <required>False</required>
      <title>IRC Handle</title>
    </field>
    <field name="image" type="plone.namedfile.field.NamedBlobImage">
      <required>False</required>
      <title>Image</title>
    </field>
  </schema>
</model>
```

See also:

- Dexterity XML
- Model-driven types

Buildout I

In this part you will:

- Learn about Buildout

Topics covered:

- Buildout
- Recipes
- Buildout Configuration
- mr.developer

Buildout composes your application for you, according to your rules.

To compose your application you must define the eggs you need, which version, what configuration files Buildout has to generate for you, what to download and compile, and so on. Buildout downloads the eggs you requested and resolves all dependencies. You might need five different eggs, but in the end, Buildout has to install 300 eggs, all with the correct version in order to resolve all the dependencies.

Buildout does this without touching your system Python or affecting any other package. The commands created by buildout bring all the required packages into the Python environment. Each command it creates may use different libraries or even different versions of the same library.

Plone needs folders for logfiles, databases and configuration files. Buildout assembles all of this for you.

You will need a lot of functionality that Buildout does not provide out of the box, so you'll need several extensions. Some extensions provide new functionality, like mr.developer, the best way to manage your checked out sources.

Syntax

The syntax of Buildout configuration files is similar to classic ini files. You write a parameter name, an equals sign and the value. If you enter another value in the next line and indent it, Buildout understands that both values belong to the parameter name, and the parameter stores all values as a list.

A Buildout consists of multiple sections. Sections start with the section name in square brackets. Each section declares a different part of your application. As a rough analogy, your Buildout file is a cookbook with multiple recipes.

There is a special section, called *[buildout]*. This section can change the behavior of Buildout itself. The variable `parts` defines which of the existing sections should actually be used.

Recipes

Buildout itself has no idea how to install Zope. Buildout is a plugin based system, it comes with a small set of plugins to create configuration files and download eggs with their dependencies and the proper version. To install a Zope site, you need a third-party plugin. The plugins provide new recipes that you have to declare and configure in their own respective sections.

One example is the section

```
[instance]
recipe = plone.recipe.zope2instance
user = admin:admin
```

This uses the python package plone.recipe.zope2instance to create and configure the Zope 2 instance which we use to run Plone. All the lines after `recipe = xyz` are the configuration of the specified recipe.

See also:

http://www.buildout.org/en/latest/docs/recipelist.html

References

Buildout allows you to use references in the configuration. A variable declaration may not only hold the variable value, but also a reference to where to look for the variable value.

If you have a big setup with many Plone sites with minor changes between each configuration, you can generate a template configuration, and each site references everything from the template and overrides just what needs to be changed.

Even in smaller buildouts this is a useful feature. We are using collective.recipe.omelette. A very practical recipe that creates a virtual directory that eases the navigation to the source code of each egg.

The omelette recipe needs to know which eggs to reference. We want the same eggs that our instance uses, so we reference the eggs of the instance instead of repeating the whole list.

Another example: Say you create configuration files for a webserver like nginx, you can define the target port for the reverse proxy by looking it up from the zope2instance recipe.

Configuring complex systems always involves a lot of duplication of information. Using references in the buildout configuration allows you to minimize these duplications.

A real life example

Let us walk through the `buildout.cfg` for the training and look at some important variables:

```
[buildout]
extends =
    http://dist.plone.org/release/5.0.6/versions.cfg
    versions.cfg
extends-cache = extends-cache

extensions = mr.developer
# Tell mr.developer to ask before updating a checkout.
always-checkout = true
show-picked-versions = true
sources = sources

# The directory this buildout is in. Modified when using vagrant.
buildout_dir = ${buildout:directory}

# We want to checkouts these eggs directly from GitHub
auto-checkout =
    ploneconf.site
#    starzel.votable_behavior

parts =
    checkversions
    codeintel
    instance
    mrbob
    packages
    robot
    test
    zopepy

eggs =
    Plone
    Pillow

# development tools
    z3c.jbot
    plone.reload
    Products.PDBDebugMode
    plone.app.debugtoolbar
    Products.PrintingMailHost

# TTW Forms (based on Archetypes)
    Products.PloneFormGen

# The addon we develop in the training
    ploneconf.site

# Voting on content
#    starzel.votable_behavior

zcml =

test-eggs +=
    ploneconf.site [test]

[instance]
recipe = plone.recipe.zope2instance
user = admin:admin
http-address = 8080
debug-mode = on
verbose-security = on
deprecation-warnings = on
eggs = ${buildout:eggs}
zcml = ${buildout:zcml}
file-storage = ${buildout:buildout_dir}/var/filestorage/Data.fs
blob-storage = ${buildout:buildout_dir}/var/blobstorage

[test]
recipe = zc.recipe.testrunner
eggs = ${buildout:test-eggs}
defaults = ['--auto-color', '-vvv']

[robot]
recipe = zc.recipe.egg
eggs =
    ${buildout:test-eggs}
    Pillow
    plone.app.robotframework[ride,reload,debug]
```

```
[packages]
recipe = collective.recipe.omelette
eggs = ${buildout:eggs}
location = ${buildout:buildout_dir}/packages

[codeintel]
recipe = corneti.recipes.codeintel
eggs = ${buildout:eggs}

[checkversions]
recipe = zc.recipe.egg
eggs = z3c.checkversions [buildout]

[zopepy]
recipe = zc.recipe.egg
eggs = ${buildout:eggs}
interpreter = zopepy
scripts =
    zopepy
    plone-generate-gruntfile
    plone-compile-resources

[mrbob]
recipe = zc.recipe.egg
eggs =
    mr.bob
    bobtemplates.plone

[sources]
ploneconf.site = git https://github.com/collective/ploneconf.site.git␣
↪pushurl=git@github.com:collective/ploneconf.site.git
starzel.votable_behavior = git https://github.com/collective/starzel.votable_behavior.
↪git pushurl=git://github.com/collective/starzel.votable_behavior.git
```

When you run **./bin/buildout** without any arguments, Buildout will look for this file.

Let us look closer at some variables.

```
extends =
    http://dist.plone.org/release/5.0.6/versions.cfg
```

This line tells Buildout to read another configuration file. You can refer to configuration files on your computer or to configuration files on the Internet, reachable via http. You can use multiple configuration files to share configurations between multiple Buildouts, or to separate different aspects of your configuration into different files. Typical examples are version specifications, or configurations that differ between different environments.

```
eggs =
    Plone
    Pillow

# development tools
    z3c.jbot
    plone.reload
    Products.PDBDebugMode
    plone.app.debugtoolbar
    Products.PrintingMailHost

# TTW Forms (based on Archetypes)
```

```
    Products.PloneFormGen

# The addon we develop in the training
    ploneconf.site

# Voting on content
#    starzel.votable_behavior

zcml =

test-eggs +=
    ploneconf.site [test]
```

This is the list of eggs that we configure to be available for Zope. These eggs are put in the python path of the script **bin/instance** with which we start and stop Plone.

The egg `Plone` is a wrapper without code. Among its dependencies is `Products.CMFPlone` which is the egg that is at the center of Plone.

The rest are add-ons we already used or will use later. The last eggs are commented out so they will not be installed by Buildout.

The file `versions.cfg` that is included by the `extends = ...` statement holds the version pins:

```
[versions]
# dev tools
mr.developer = 1.34
Products.PDBDebugMode = 1.3.1
corneti.recipes.codeintel = 0.3
plone.app.debugtoolbar = 1.1.1
z3c.jbot = 0.7.2
Products.PrintingMailHost = 1.0

# pins for some Addons
Products.PloneFormGen = 1.8.1
Products.PythonField = 1.1.3
# ...
```

This is another special section. By default buildout will look for version pins in a section called `[versions]`. This is why we included the file `versions.cfg`.

Hello mr.developer!

There are many more important things to know, and we can't go through them all in detail but I want to focus on one specific feature: `mr.developer`

With `mr.developer` you can declare which packages you want to check out from which version control system and which repository URL. You can check out sources from git, svn, bzr, hg and maybe more. Also, you can say that some sources are in your local file system.

`mr.developer` comes with a command, **./bin/develop**. You can use it to update your code, to check for changes and so on. You can activate and deactivate your source checkouts. If you develop your extensions in eggs with separate checkouts, which is a good practice, you can plan releases by having all source checkouts deactivated, and only activate them when you write changes that require a new release. You can activate and deactivate eggs via the **develop** command or the Buildout configuration. You should always use the Buildout way. Your commit serves as documentation.

Extensible

You might have noticed that most if not all functionality is only available via plugins. One of the things that Buildout excels at without any plugin is the dependency resolution. You can help Plone in dependency resolution by declaring exactly which version of an egg you want. This is only one use case. Another one is much more important: If you want to have a repeatable Buildout, one that works two months from now also, you *must* declare all your egg versions. Else Buildout might install newer versions.

Be McGuyver

As you can see, you can build very complex systems with Buildout. It is time for some warnings. Be selective in your recipes. Supervisor is a program to manage running servers, and it's pretty good. There is a recipe for it.

The configuration for this recipe is more complicated than the supervisor configuration itself! By using this recipe, you force others to understand the recipe's specific configuration syntax *and* the supervisor syntax. For such cases, collective.recipe.template is a better match.

Another problem is error handling. Buildout tries to install a weird dependency you do not actually want? Buildout will not tell you where it is coming from.

If there is a problem, you can always run Buildout with `-v` to get more verbose output, sometimes it helps.

```
$ ./bin/buildout -v
```

If strange egg versions are requested, check the dependencies declaration of your eggs and your version pinnings. Here is an invaluable shell command that allows you to find all packages that depend on a particular egg and version:

```
$ grep your.egg.name.here /home/vagrant/buildout-cache/eggs/*.egg/EGG-INFO/requires.
↪txt
```

Put the name of the egg with a version conflict as the first argument. Also, change the path to the buildout cache folder according to your installation (the vagrant buildout is assumed in the example).

Some parts of Buildout interpret egg names case sensitively, others don't. This can result in funny problems.

Always check out the ordering of your extends, always use the `annotate` command of Buildout to see if it interprets your configuration differently than you. Restrict yourself to simple Buildout files. You can reference variables from other sections, you can even use a whole section as a template. We learned that this does not work well with complex hierarchies and had to abandon that feature.

In the chapter *Buildout II: Getting Ready for Deployment* we will have a look at a production-ready buildout for Plone that has many useful features.

See also:

Buildout-Documentation http://docs.buildout.org/en/latest/contents.html

Troubleshooting http://docs.plone.org/manage/troubleshooting/buildout.html

A minimal buildout for Plone 5 https://github.com/collective/minimalplone5

A minimal buildout for Plone 4 https://github.com/collective/minimalplone4

The buildout of the unified installer has some valuable documentation as inline-comment

- https://github.com/plone/Installers-UnifiedInstaller/blob/master/buildout_templates/buildout.cfg
- https://github.com/plone/Installers-UnifiedInstaller/blob/master/base_skeleton/base.cfg
- https://github.com/plone/Installers-UnifiedInstaller/blob/master/base_skeleton/develop.cfg

mr.developer https://pypi.python.org/pypi/mr.developer/

Write Your Own Add-Ons to Customize Plone

Get the code!

Get the code for this chapter (More info):

```
git checkout eggs1
```

In this part you will:

- Create a custom Python package `ploneconf.site` to hold all the code
- Modify buildout to install that package

Topics covered:

- `mr.bob` and `bobtemplates.plone`
- the structure of eggs

Creating the package

Our own code has to be organized as a Python package, also called *egg*. An egg is a zip file or a directory that follows certain conventions. We are going to use bobtemplates.plone to create a skeleton project. We only need to fill in the blanks.

We create and enter the `src` directory (*src* is short for *sources*) and call a script called **`mrbob`** from our buildout's `bin` directory:

```
$ mkdir src      # (if src does not exist already)
$ cd src
$ ../bin/mrbob -O ploneconf.site bobtemplates:plone_addon
```

We have to answer some questions about the add-on. We will press `Enter` (i.e. choosing the default value) for all questions except 3 (where you enter your GitHub username if you have one) and 5 (Plone version), where we enter `5.0.6`:

```
--> What kind of package would you like to create? Choose between 'Basic', 'Dexterity
↪', and 'Theme'. [Basic]:

--> Author's name [Philip Bauer]:

--> Author's email [bauer@starzel.de]:

--> Author's GitHub username: fulv

--> Package description [An add-on for Plone]:

--> Plone version [5.0.5]: 5.0.6

Generated file structure at /vagrant/buildout/src/ploneconf.site
```

If this is your first egg, this is a very special moment. We are going to create the egg with a script that generates a lot of necessary files. They all are necessary, but sometimes in a subtle way. It takes a while to understand their full meaning. Only last year I learned and understood why I should have a `MANIFEST.in` file. You can get along without one, but trust me, you get along better with a proper manifest file.

Inspecting the package

In `src` there is now a new folder `ploneconf.site` and in there is the new package. Let's have a look at some of the files:

bootstrap-buildout.py, buildout.cfg, travis.cfg, .travis.yml, .coveragerc You can ignore these files for now. They are here to create a buildout only for this egg to make testing it easier. Once we start writing tests for this package we will have another look at them.

README.rst, CHANGES.rst, CONTRIBUTORS.rst, docs/ The documentation, changelog, the list of contributors and the license of your egg goes in here.

setup.py This file configures the package, its name, dependencies and some metadata like the author's name and email address. The dependencies listed here are automatically downloaded when running buildout.

src/ploneconf/site/ The package itself lives inside a special folder structure. That seems confusing but is necessary for good testability. Our package contains a namespace package called *ploneconf.site* and because of this there is a folder `ploneconf` with a `__init__.py` and in there another folder `site` and in there finally is our code. From the buildout's perspective our code is in *your buildout directory*/src/ploneconf.site/src/ploneconf/site/*real code*

Note: Unless discussing the buildout we will from now on silently omit these folders when describing files and assume that *your buildout directory*/src/ploneconf.site/src/ploneconf/site/ is the root of our package!

configure.zcml (src/ploneconf/site/configure.zcml) The phone book of the distribution. By reading it you can find out which functionality is registered using the component architecture.

setuphandlers.py (src/ploneconf/site/setuphandlers.py) This holds code that is automatically run when installing and uninstalling our add-on.

interfaces.py (src/ploneconf/site/interfaces.py) Here a browserlayer is defined in a straightforward python class. We will need it later.

testing.py This holds the setup for running tests.

tests/ This holds the tests.

browser/ This directory is a python package (because it has a `__init__.py`) and will by convention hold most things that are visible in the browser.

browser/configure.zcml The phonebook of the browser package. Here views, resources and overrides are registered.

browser/overrides/ This add-on is already configured to allow overriding existing default Plone templates.

browser/static/ A directory that holds static resources (images/css/js). The files in here will be accessible through URLs like `++resource++ploneconf.site/myawesome.css`

profiles/default/ This folder contains the GenericSetup profile. During the training we will put some XML files here that hold configuration for the site.

profiles/default/metadata.xml Version number and dependencies that are auto-installed when installing our add-on.

Including the package in Plone

Before we can use our new package we have to tell Plone about it. Look at `buildout.cfg` and see how `ploneconf.site` is included in *auto-checkout*, *eggs* and *test*:

```
auto-checkout +=
    ploneconf.site
#    starzel.votable_behavior

parts =
    checkversions
    codeintel
    instance
    mrbob
    packages
    robot
    test
    zopepy

eggs =
    Plone
    Pillow

# development tools
    z3c.jbot
    plone.api
    plone.reload
    Products.PDBDebugMode
    plone.app.debugtoolbar
    Products.PrintingMailHost

# TTW Forms (based on Archetypes)
    Products.PloneFormGen

# The add-on we develop in the training
    ploneconf.site

# Voting on content
#    starzel.votable_behavior

zcml =

test-eggs +=
    ploneconf.site [test]
```

This tells Buildout to add the egg `ploneconf.site`. The sources for this eggs are defined in the section `[sources]` at the bottom of `buildout.cfg`.

```
[sources]
ploneconf.site = git https://github.com/collective/ploneconf.site.git pushurl=git@github.com:collective/ploneconf.site.git
starzel.votable_behavior = git https://github.com/collective/starzel.votable_behavior.git pushurl=git://github.com/collective/starzel.votable_behavior.git
```

This tells buildout not to download it from pypi but to do a checkout from GitHub put the code in `src/ploneconf.site`.

Note: The package `ploneconf.site` is now downloaded from GitHub and automatically in the branch master

Note: If you do **not** want to use the prepared package for ploneconf.site from GitHub but write it yourself (we suggest you try that) then add the following instead:

```
[sources]
ploneconf.site = fs ploneconf.site path=src
starzel.votable_behavior = git https://github.com/collective/starzel.votable_behavior.
↪git pushurl=git://github.com/collective/starzel.votable_behavior.git
```

This tells buildout to expect *ploneconf.site* in `src/ploneconf.site`. The directive `fs` allows you to add eggs on the filesystem without a version control system.

Now run buildout to reconfigure Plone with the updated configuration:

```
$ ./bin/buildout
```

After restarting Plone with **`./bin/instance fg`** the new add-on `ploneconf.site` is available for install like PloneFormGen or Plone True Gallery.

We will not install it now since we did not add any of our own code or configuration yet. Let's do that next.

Return to Dexterity: Moving contenttypes into Code

Get the code!

Get the code for this chapter (More info):

```
git checkout export_code
```

In this part you will:

- Move the *Talk* type into `ploneconf.site`
- Improve the schema and the FTI

Topics covered:

- FTI
- type definitions with generic setup
- XML schema
- more widgets

Remember the *Talk* content type that we created through-the-web with Dexterity? Let's move that new content type into our add-on package so that it may be installed in other sites without TTW manipulation.

Steps:

- Return to the Dexterity control panel
- Export the *Talk* Type Profile and save the file
- Delete the *Talk* from the site before installing it from the file system
- Extract the files from the exported tar file and add them to our add-on package in `profiles/default/`

Note: From the buildout directory perspective that is `src/ploneconf.site/src/ploneconf/site/profiles/default/`

The file `profiles/default/types.xml` tells Plone that there is a new content type defined in file `talk.xml`.

```xml
<?xml version="1.0"?>
<object name="portal_types" meta_type="Plone Types Tool">
 <property name="title">Controls the available contenttypes in your portal</property>
 <object name="talk" meta_type="Dexterity FTI"/>
 <!-- -*- more types can be added here -*- -->
</object>
```

Upon installing, Plone reads the file `profiles/default/types/talk.xml` and registers a new type in `portal_types` (you can find and inspect this tool in the ZMI!) with the information taken from that file.

```xml
  <?xml version="1.0"?>
  <object name="talk" meta_type="Dexterity FTI" i18n:domain="plone"
     xmlns:i18n="http://xml.zope.org/namespaces/i18n">
   <property name="title" i18n:translate="">Talk</property>
   <property name="description" i18n:translate="">None</property>
   <property name="icon_expr">string:${portal_url}/document_icon.png</property>
   <property name="factory">talk</property>
   <property name="add_view_expr">string:${folder_url}/++add++talk</property>
   <property name="link_target"></property>
   <property name="immediate_view">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">view</property>
   <property name="view_methods">
    <element value="view"/>
   </property>
   <property name="default_view_fallback">False</property>
   <property name="add_permission">cmf.AddPortalContent</property>
   <property name="klass">plone.dexterity.content.Container</property>
   <property name="behaviors">
    <element value="plone.app.dexterity.behaviors.metadata.IDublinCore"/>
    <element value="plone.app.content.interfaces.INameFromTitle"/>
   </property>
   <property name="schema"></property>
   <property
      name="model_source">&lt;?xml version='1.0' encoding='utf8'?&gt;
&lt;model xmlns:lingua="http://namespaces.plone.org/supermodel/lingua" xmlns:users=
↪"http://namespaces.plone.org/supermodel/users" xmlns:form="http://namespaces.plone.
↪org/supermodel/form" xmlns:i18n="http://xml.zope.org/namespaces/i18n" xmlns:
↪security="http://namespaces.plone.org/supermodel/security" xmlns:marshal="http://
↪namespaces.plone.org/supermodel/marshal" xmlns="http://namespaces.plone.org/
↪supermodel/schema"&gt;
      &lt;schema&gt;
        &lt;field name="type_of_talk" type="zope.schema.Choice"&gt;
          &lt;description/&gt;
          &lt;title&gt;Type of talk&lt;/title&gt;
          &lt;values&gt;
            &lt;element&gt;Talk&lt;/element&gt;
            &lt;element&gt;Training&lt;/element&gt;
            &lt;element&gt;Keynote&lt;/element&gt;
          &lt;/values&gt;
        &lt;/field&gt;
        &lt;field name="details" type="plone.app.textfield.RichText"&gt;
          &lt;description&gt;Add a short description of the talk (max. 2000␣
↪characters)&lt;/description&gt;
```

```
        &lt;max_length&gt;2000&lt;/max_length&gt;
        &lt;title&gt;Details&lt;/title&gt;
      &lt;/field&gt;
      &lt;field name="audience" type="zope.schema.Set"&gt;
        &lt;description/&gt;
        &lt;title&gt;Audience&lt;/title&gt;
        &lt;value_type type="zope.schema.Choice"&gt;
          &lt;values&gt;
            &lt;element&gt;Beginner&lt;/element&gt;
            &lt;element&gt;Advanced&lt;/element&gt;
            &lt;element&gt;Professionals&lt;/element&gt;
          &lt;/values&gt;
        &lt;/value_type&gt;
      &lt;/field&gt;
      &lt;field name="speaker" type="zope.schema.TextLine"&gt;
        &lt;description&gt;Name (or names) of the speaker&lt;/description&gt;
        &lt;title&gt;Speaker&lt;/title&gt;
      &lt;/field&gt;
      &lt;field name="email" type="plone.schema.email.Email"&gt;
        &lt;description&gt;Adress of the speaker&lt;/description&gt;
        &lt;title&gt;Email&lt;/title&gt;
      &lt;/field&gt;
      &lt;field name="image" type="plone.namedfile.field.NamedBlobImage"&gt;
        &lt;description/&gt;
        &lt;required&gt;False&lt;/required&gt;
        &lt;title&gt;Image&lt;/title&gt;
      &lt;/field&gt;
      &lt;field name="speaker_biography" type="plone.app.textfield.RichText"&gt;
        &lt;description/&gt;
        &lt;max_length&gt;1000&lt;/max_length&gt;
        &lt;required&gt;False&lt;/required&gt;
        &lt;title&gt;Speaker Biography&lt;/title&gt;
      &lt;/field&gt;
    &lt;/schema&gt;
  &lt;/model&gt;</property>
 <property name="model_file"></property>
 <property name="schema_policy">dexterity</property>
 <alias from="(Default)" to="(dynamic view)"/>
 <alias from="edit" to="@@edit"/>
 <alias from="sharing" to="@@sharing"/>
 <alias from="view" to="(selected layout)"/>
 <action title="View" action_id="view" category="object" condition_expr=""
    description="" icon_expr="" link_target="" url_expr="string:${object_url}"
    visible="True">
  <permission value="View"/>
 </action>
 <action title="Edit" action_id="edit" category="object" condition_expr=""
    description="" icon_expr="" link_target=""
    url_expr="string:${object_url}/edit" visible="True">
  <permission value="Modify portal content"/>
 </action>
</object>
```

Now our package has some real contents. So, we’ll need to reinstall it (if installed before).

- Restart Plone.
- Re-install ploneconf.site (deactivate and activate).

- Test the type by adding an object or editing one of the old ones.
- Look at how the talks are presented in the browser.

The escaped inline xml is simply too ugly to look at. You should move it to a separate file!

Create a new folder `content` in the main directory (from the buildout directory perspective that is `src/ploneconf.site/src/ploneconf/site/content/`). Inside add an empty file `__init__py` and a file `talk.xml` that contains the real XML (copied from http://localhost:8080/Plone/dexterity-types/talk/@@modeleditor and beautified with some online XML formatter (http://lmgtfy.com/?q=xml+formatter))

```
<?xml version='1.0' encoding='utf8'?>
  <model xmlns="http://namespaces.plone.org/supermodel/schema"
         xmlns:form="http://namespaces.plone.org/supermodel/form"
         xmlns:i18n="http://xml.zope.org/namespaces/i18n"
         xmlns:lingua="http://namespaces.plone.org/supermodel/lingua"
         xmlns:marshal="http://namespaces.plone.org/supermodel/marshal"
         xmlns:security="http://namespaces.plone.org/supermodel/security"
         xmlns:users="http://namespaces.plone.org/supermodel/users">
    <schema>
      <field name="type_of_talk" type="zope.schema.Choice">
        <description/>
        <title>Type of Talk</title>
        <values>
          <element>Talk</element>
          <element>Training</element>
          <element>Keynote</element>
        </values>
      </field>
      <field name="details" type="plone.app.textfield.RichText">
        <description>Add a short description of the talk (max. 2000 characters)</description>/&gt;
        <max_length>2000</max_length>
        <title>Details</title>
      </field>
      <field name="audience" type="zope.schema.Set">
        <description/>
        <title>Audience</title>
        <value_type type="zope.schema.Choice">
          <values>
            <element>Beginner</element>
            <element>Advanced</element>
            <element>Professional</element>
          </values>
        </value_type>
      </field>
      <field name="speaker" type="zope.schema.TextLine">
        <description>Name (or names) of the speaker</description>/&gt;
        <title>Speaker</title>
      </field>
      <field name="email" type="plone.schema.email.Email">
        <description>Adress of the speaker</description>/&gt;
        <title>Email</title>
      </field>
      <field name="image" type="plone.namedfile.field.NamedBlobImage">
        <description/>
        <required>False</required>
        <title>Image</title>
      </field>
      <field name="speaker_biography" type="plone.app.textfield.RichText">
        <description/>
        <max_length>1000</max_length>
        <required>False</required>
        <title>Speaker Biography</title>
      </field>
    </schema>
  </model>
```

Now remove the ugly model_source and instead point to the new XML file in the FTI by using the property `model_file`:

```
<property name="model_source"></property>
<property name="model_file">ploneconf.site.content:talk.xml</property>
```

`ploneconf.site.content:talk.xml` points to a file `talk.xml` to be found in the Python path `ploneconf.site.content`. The `__init__.py` is needed to turn the folder `content` into a Python package. It is best-practice to add schemas in this folder, and in later chapters you will add new types with pythons-schemata in the same folder.

Note: The default types of Plone 5 also have an xml schema like this since that allows the fields of the types to be editable trough the web! Fields for types with a python schema are not editable ttw.

Changing a widget

Dexterity XML is very powerful. By editing it (not all features have a UI) you should be able to do everything you can do with a Python schema. Sadly not every feature also is exposed in the UI of the dexterity schema editor. For example you cannot yet change the widgets or permissions for fields in the UI. We need to do this in the xml- or python-schema.

Our talks use a dropdown for *type_of_talk* and a multiselect for *audience*. Radio-buttons and checkboxes would be the better choice here. Modify the XML to make that change happen:

```
<?xml version="1.0" encoding="UTF-8"?>
<model xmlns="http://namespaces.plone.org/supermodel/schema"
       xmlns:form="http://namespaces.plone.org/supermodel/form"
       xmlns:i18n="http://xml.zope.org/namespaces/i18n"
       xmlns:lingua="http://namespaces.plone.org/supermodel/lingua"
       xmlns:marshal="http://namespaces.plone.org/supermodel/marshal"
       xmlns:security="http://namespaces.plone.org/supermodel/security"
       xmlns:users="http://namespaces.plone.org/supermodel/users">
  <schema>
    <field name="type_of_talk" type="zope.schema.Choice"
      form:widget="z3c.form.browser.radio.RadioFieldWidget">
      <description />
      <title>Type of talk</title>
      <values>
        <element>Talk</element>
        <element>Training</element>
        <element>Keynote</element>
      </values>
    </field>
    <field name="details" type="plone.app.textfield.RichText">
      <description>Add a short description of the talk (max. 2000 characters)</
↪description>
```

```
      <max_length>2000</max_length>
      <title>Details</title>
    </field>
    <field name="audience" type="zope.schema.Set"
      form:widget="z3c.form.browser.checkbox.CheckBoxFieldWidget">
      <description />
      <title>Audience</title>
      <value_type type="zope.schema.Choice">
        <values>
          <element>Beginner</element>
          <element>Advanced</element>
          <element>Professionals</element>
        </values>
      </value_type>
    </field>
    <field name="speaker" type="zope.schema.TextLine">
      <description>Name (or names) of the speaker</description>
      <title>Speaker</title>
    </field>
    <field name="email" type="plone.schema.email.Email">
      <description>Adress of the speaker</description>
      <title>Email</title>
    </field>
    <field name="image" type="plone.namedfile.field.NamedBlobImage">
      <description />
      <required>False</required>
      <title>Image</title>
    </field>
    <field name="speaker_biography" type="plone.app.textfield.RichText">
      <description />
      <max_length>1000</max_length>
      <required>False</required>
      <title>Speaker Biography</title>
    </field>
  </schema>
</model>
```

Protect fields with permissions

We also want to have a add a new field *room* to show where a talk will take place. Our case-study says the speakers will submit the talks online. How should they know in which room the talk will take place (if it got accepted at all)? So we need to hide this field from them by requiring a permission that they do not have.

Let's assume the prospective speakers will not have the permission to review content (i.e. edit submitted content and publish it) but the organizing commitee has. You can then protect the field using the permission *Review portal content* in this case the name of the permission-utility for this permission: *cmf.ReviewPortalContent*.

We only want to prevent writing, not reading, so we'll only manage the *write-permission*:

```
<?xml version="1.0" encoding="UTF-8"?>
<model xmlns="http://namespaces.plone.org/supermodel/schema"
       xmlns:form="http://namespaces.plone.org/supermodel/form"
       xmlns:i18n="http://xml.zope.org/namespaces/i18n"
       xmlns:lingua="http://namespaces.plone.org/supermodel/lingua"
       xmlns:marshal="http://namespaces.plone.org/supermodel/marshal"
       xmlns:security="http://namespaces.plone.org/supermodel/security"
       xmlns:users="http://namespaces.plone.org/supermodel/users">
```

```
    <schema>
      <field name="type_of_talk" type="zope.schema.Choice"
        form:widget="z3c.form.browser.radio.RadioFieldWidget">
        <description />
        <title>Type of talk</title>
        <values>
          <element>Talk</element>
          <element>Training</element>
          <element>Keynote</element>
        </values>
      </field>
      <field name="details" type="plone.app.textfield.RichText">
        <description>Add a short description of the talk (max. 2000 characters)</description>
        <max_length>2000</max_length>
        <title>Details</title>
      </field>
      <field name="audience"
             type="zope.schema.Set"
             form:widget="z3c.form.browser.checkbox.CheckBoxFieldWidget">
        <description />
        <title>Audience</title>
        <value_type type="zope.schema.Choice">
          <values>
            <element>Beginner</element>
            <element>Advanced</element>
            <element>Professionals</element>
          </values>
        </value_type>
      </field>
      <field name="room"
             type="zope.schema.Choice"
             form:widget="z3c.form.browser.radio.RadioFieldWidget"
             security:write-permission="cmf.ReviewPortalContent">
        <description></description>
        <required>False</required>
        <title>Room</title>
        <values>
          <element>101</element>
          <element>201</element>
          <element>Auditorium</element>
        </values>
      </field>
      <field name="speaker" type="zope.schema.TextLine">
        <description>Name (or names) of the speaker</description>
        <title>Speaker</title>
      </field>
      <field name="email" type="plone.schema.email.Email">
        <description>Adress of the speaker</description>
        <title>Email</title>
      </field>
      <field name="image" type="plone.namedfile.field.NamedBlobImage">
        <description />
        <required>False</required>
        <title>Image</title>
      </field>
      <field name="speaker_biography" type="plone.app.textfield.RichText">
        <description />
        <max_length>1000</max_length>
        <required>False</required>
        <title>Speaker Biography</title>
      </field>
    </schema>
  </model>
```

See also:

- http://docs.plone.org/external/plone.app.dexterity/docs/reference/dexterity-xml.html
- https://github.com/plone/plone.autoform/blob/master/plone/autoform/supermodel.txt

Exercise 1

Create a new package called `collective.behavior.myfeature`. Inspect the directory structure of this package. Delete it after you are done.

Solution

```
$ cd src
$ ../bin/mrbob -O collective.behavior.myfeature bobtemplates:plone_addon
```

Many packages that are part of Plone and some add-ons use a nested namespace such as `plone.app.contenttypes`.

Exercise 2

Go to the ZMI and look for the definition of the new `Talk` content type in `portal_types`. Now deactivate *Implicitly addable?* and save. Go back to the site. Can you identify what this change has caused? And why is that useful?

Solution

Go to http://localhost:8080/Plone/portal_types/talk/manage_propertiesForm

When disabling *Implicitly addable* you can no longer add Talks any more unless you change some container like the type *Folder*: Enable *Filter contenttypes?* for it and add *Talk* to the items that are allowed.

With this method you can prevent content that only makes sense inside some defined structure to show up in places where they do not belong.

The equivalent setting for disabling *Implicitly addable* in `Talk.xml` is:

```
<property name="global_allow">False</property>
```

Views I

Get the code!

Get the code for this chapter (More info):

```
git checkout views_1
```

In this part you will:

- Register a view
- Create and use a template for the view

Topics covered:

- zcml

A simple browser view

Before writing the talk view itself we step back and talk *a little* about views and templates.

A view in Plone is usually a `BrowserView`. It can hold a lot of cool python code but we will first focus on the template.

Edit the file `browser/configure.zcml` and register a new view called *training*:

```
<configure
  xmlns="http://namespaces.zope.org/zope"
  xmlns:browser="http://namespaces.zope.org/browser"
  xmlns:plone="http://namespaces.plone.org/plone"
  i18n_domain="ploneconf.site">

  <!-- Set overrides folder for Just-a-Bunch-Of-Templates product -->
  <include package="z3c.jbot" file="meta.zcml" />
  <browser:jbot
    directory="overrides"
    layer="ploneconf.site.interfaces.IPloneconfSiteLayer"
    />

  <!-- Publish static files -->
  <browser:resourceDirectory
    name="ploneconf.site"
    directory="static"
    />

  <browser:page
    name="training"
    for="*"
    template="templates/training.pt"
    permission="zope2.View"
    />

</configure>
```

Add a file `browser/templates/training.pt`

```
<h1>Hello World</h1>
```

- Restart Plone and open http://localhost:8080/Plone/@@training.

- You should now see “Hello World”.

We now have everything in place to learn about page templates.

Note: The view `training` has no python class registered for it but only a template. It acts as if it had an empty python class inheriting from `Products.Five.browser.BrowserView` but the way that happens is actually quite a bit of magic...

Page Templates

Get the code!

Get the code for this chapter (More info):

```
git checkout zpt
```

In this part you will:

- Learn to write page templates

Topics covered:

- TAL and TALES
- METAL
- Chameleon

Page Templates are HTML files with some additional information, written in TAL, METAL and TALES.

Page templates must be valid xml.

The three languages are:

- TAL: “Template Attribute Language”
  - Templating XML/HTML using special attributes
  - Using TAL we include expressions within html
- TALES: “TAL Expression Syntax”
  - defines the syntax and semantics of these expressions
- METAL: “Macro Expansion for TAL”
  - this enables us to combine, re-use and nest templates together

TAL and METAL are written like html attributes (href, src, title). TALES are written like the values of html attributes. A typical TAL attribute looks like this:

```
<title tal:content="context/title">
    The Title of the content
</title>
```

It’s used to modify the output:

```
<p tal:content="string:I love red">I love blue</p>
```

results in:

```
<p>I love red</p>
```

Let's try it. Open the file `training.pt` and add:

```
<html>
<body>

    <p>red</p>

</body>
</html>
```

TAL and TALES

Let's add some magic and modify the <p>-tag:

```
<p tal:content="string:blue">red</p>
```

This will result in:

```
<p>blue</p>
```

Without restarting Plone open http://localhost:8080/Plone/@@training.

The same happens with attributes. Replace the <p>-line with:

```
<a href="http://www.mssharepointconference.com"
   tal:define="a_fine_url string:http://www.ploneconf.org"
   tal:attributes="href a_fine_url"
   tal:content="string:An even better conference">
    A sharepoint conference
</a>
```

results in:

```
<a href="http://www.ploneconf.org">
    An even better conference
</a>
```

We used three TAL-Attributes here. This is the complete list of TAL-attributes:

tal:define define variables. We defined the variable `a_fine_url` to the string "http://www.ploneconf.org"

tal:content replace the content of an element. We replaced the default content above with "An even better conference"

tal:attributes dynamically change element attributes. We set the HTML attribute `href` to the value of the variable `a_fine_url`

tal:condition tests whether the expression is true or false, and outputs or omits the element accordingly.

tal:repeat repeats an iterable element, in our case the list of talks.

tal:replace replace the content of an element, like `tal:content` does, but removes the element only leaving the content.

tal:omit-tag remove an element, leaving the content of the element.

tal:on-error handle errors.

python expressions

So far we only used one TALES expression (the `string:` bit). Let's use a different TALES expression now. With `python:` we can use python code. A simple example:

```
<p tal:define="title context/title"
   tal:content="python:title.upper()">
   A big title
</p>
```

And another:

```
<p tal:define="talks python:['Dexterity for the win!',
                             'Deco is the future',
                             'A keynote on some weird topic',
                             'The talk that I did not submit']"
   tal:content="python:talks[0]">
    A talk
</p>
```

With python expressions

- you can only write single statements
- you could import things but you should not (example: `tal:define="something modules/Products.PythonScripts/something;`).

tal:condition

tal:condition tests whether the expression is true or false.

- If it's true, then the tag is rendered.
- If it's false then the tag **and all its children** are removed and no longer evaluated.
- We can reverse the logic by prepending a `not:` to the expression.

Let's add another TAL Attribute to our above example:

```
tal:condition="talks"
```

We could also test for the number of talks:

```
tal:condition="python:len(talks) >= 1"
```

or if a certain talk is in the list of talks:

```
tal:condition="python:'Deco is the future' in talks"
```

tal:repeat

Let's try another attribute:

```
<p tal:define="talks python:['Dexterity for the win!',
                             'Deco is the future',
                             'A keynote on some weird topic',
                             'The talk that I did not submit']"
   tal:repeat="talk talks"
   tal:content="talk">
   A talk
</p>
```

`tal:repeat` repeats an iterable element, in our case the list of talks.

We change the markup a little to construct a list in which there is an `<li>` for every talk:

```
<ul tal:define="talks python:['Dexterity for the win!',
                              'Deco is the future',
                              'A keynote on some weird topic',
                              'The talk that I did not submit']">
    <li tal:repeat="talk talks"
        tal:content="talk">
          A talk
    </li>
    <li tal:condition="not:talks">
          Sorry, no talks yet.
    </li>
</ul>
```

path expressions

Regarding TALES so far we used `string:` or `python:` or only variables. The next and most common expression are path expressions. Optionally you can start a path expression with `path:`

Every path expression starts with a variable name. It can either be an object like `context`, `view` or `template` or a variable defined earlier like `talk`.

After the variable we add a slash `/` and the name of a sub-object, attribute or callable. The `/` is used to end the name of an object and the start of the property name. Properties themselves may be objects that in turn have properties.

```
<p tal:content="context/title"></p>
```

We can chain several of those to get to the information we want.

```
<p tal:content="context/REQUEST/form"></p>
```

This would return the value of the form dictionary of the HTTPRequest object. Useful for form handling.

The `|` (“or”) character is used to find an alternative value to a path if the first path evaluates to `nothing` or does not exist.

```
<p tal:content="context/title | context/id"></p>
```

This returns the id of the context if it has no title.

```
<p tal:replace="talk/average_rating | nothing"></p>
```

This returns nothing if there is no ‘average_rating’ for a talk. What will not work is `tal:content="python:talk['average_rating'] or ''"`. Who knows what this would yield?

We'll get `KeyError: 'average_rating'`. It is very bad practice to use `|` too often since it will swallow errors like a typo in `tal:content="talk/averange_ratting | nothing"` and you might wonder why there are no ratings later on...

You can't and should not use it to prevent errors like a try/except block.

There are several **built-in variables** that can be used in paths:

The most frequently used one is `nothing` which is the equivalent to None

```
<p tal:replace="nothing">
    this comment will not be rendered
</p>
```

A dict of all the available variables is `econtext`

```
<dl tal:define="path_variables_dict econtext">
  <tal:vars tal:repeat="variable path_variables_dict">
    <dt>${variable}</dt>
    <dd>${python:path_variables_dict[variable]}</dd>
  </tal:vars>
</dl>
```

Note: In Plone 4 that used to be `CONTEXTS`

```
<dl tal:define="path_variables_dict CONTEXTS">
  <tal:vars tal:repeat="variable path_variables_dict">
    <dt tal:content="variable"></dt>
    <dd tal:content="python:path_variables_dict[variable]"></dd>
  </tal:vars>
</dl>
```

Useful for debugging :-)

Pure TAL blocks

We can use TAL attributes without HTML Tags. This is useful when we don't need to add any tags to the markup

Syntax:

```
<tal:block attribute="expression">some content</tal:block>
```

Examples:

```
<tal:block define="id template/id">
...
  <b tal:content="id">The id of the template</b>
...
</tal:block>

<tal:news condition="python:context.portal_type == 'News Item'">
    This text is only visible if the context is a News Item
</tal:news>
```

handling complex data in templates

Let’s move on to a little more complex data. And to another TAL attribute:

tal:replace replace the content of an element and removes the element only leaving the content.

Example:

```
<p>
    <img tal:define="tag string:<img src='https://plone.org/logo.png'>"
         tal:replace="tag">
</p>
```

this results in:

```
<p>
    &lt;img src='https://plone.org/logo.png'&gt;
</p>
```

`tal:replace` drops its own base tag in favor of the result of the TALES expression. Thus the original `<img... >` is replaced. But the result is escaped by default.

To prevent escaping we use `structure`

```
<p>
    <img tal:define="tag string:<img src='https://plone.org/logo.png'>"
         tal:replace="structure tag">
</p>
```

Now let’s emulate a typical Plone structure by creating a dictionary.

```
<table tal:define="talks python:[{'title':'Dexterity for the win!',
                                  'subjects':('content-types', 'dexterity')},
                                 {'title':'Deco is the future',
                                  'subjects':('layout', 'deco')},
                                 {'title':'The State of Plone',
                                  'subjects':('keynote',) },
                                 {'title':'Diazo designs dont suck!',
                                  'subjects':('design', 'diazo', 'xslt')}
                                ]">
    <tr>
        <th>Title</th>
        <th>Topics</th>
    </tr>
    <tr tal:repeat="talk talks">
        <td tal:content="talk/title">A talk</td>
        <td tal:define="subjects talk/subjects">
            <span tal:repeat="subject subjects"
                  tal:replace="subject">
            </span>
        </td>
    </tr>
</table>
```

We emulate a list of talks and display information about them in a table. We’ll get back to the list of talks later when we use the real talk objects that we created with dexterity.

To complete the list here are the TAL attributes we have not yet used:

`tal:omit-tag` Omit the element tag, leaving only the inner content.

tal:on-error handle errors.

When an element has multiple TAL attributes, they are executed in this order:

1. define
2. condition
3. repeat
4. content or replace
5. attributes
6. omit-tag

Plone 5

Plone 5 uses a new rendering engine called Chameleon. Using the integration layer five.pt it is fully compatible with the normal TAL syntax but offers some additional features:

You can use `${...}` as short-hand for text insertion in pure html effectively making `tal:content` and `tal:attributes` obsolete. Here are some examples:

Plone 4 and Plone 5:

```
<a tal:attributes="href string:${context/absolute_url}?ajax_load=1;
                   class python:context.portal_type.lower().replace(' ', '')"
   tal:content="context/title">
   The Title of the current object
</a>
```

Plone 5 (and Plone 4 with five.pt) :

```
<a href="${context/absolute_url}?ajax_load=1"
   class="${python:context.portal_type.lower().replace(' ', '')}">
   ${python:context.title}
</a>
```

You can also add pure python into the templates:

```
<div>
  <?python
  someoptions = dict(
      id=context.id,
      title=context.title)
  ?>
  This object has the id "${python:someoptions['id']}"" and the title "${python:
↪someoptions['title']}".
</div>
```

Exercise 1

Modify the following template and one by one solve the following problems: :

```
<table tal:define="talks python:[{'title': 'Dexterity is the new default!',
                                  'subjects': ('content-types', 'dexterity')},
                                 {'title': 'Mosaic will be the next big thing.',
                                  'subjects': ('layout', 'deco', 'views'),
                                  'url': 'https://www.youtube.com/watch?v=QSNufxaYb1M'},
                                 {'title': 'The State of Plone',
                                  'subjects': ('keynote',) },
                                 {'title': 'Diazo is a powerful tool for theming!',
                                  'subjects': ('design', 'diazo', 'xslt')},
                                 {'title': 'Magic templates in Plone 5',
                                  'subjects': ('templates', 'TAL'),
                                  'url': 'http://www.starzel.de/blog/magic-templates-in-plone-5'}
                                ]">
    <tr>
        <th>Title</th>
        <th>Topics</th>
    </tr>
    <tr tal:repeat="talk talks">
        <td tal:content="talk/title">A talk</td>
        <td tal:define="subjects talk/subjects">
            <span tal:repeat="subject subjects"
                  tal:replace="subject">
            </span>
        </td>
    </tr>
</table>
```

1. Display the subjects as comma-separated.

Solution

```
<table tal:define="talks python:[{'title': 'Dexterity is the new default!',
                                  'subjects': ('content-types', 'dexterity')},
                                 {'title': 'Mosaic will be the next big thing.',
                                  'subjects': ('layout', 'deco', 'views'),
                                  'url': 'https://www.youtube.com/watch?v=QSNufxaYb1M'},
                                 {'title': 'The State of Plone',
                                  'subjects': ('keynote',) },
                                 {'title': 'Diazo is a powerful tool for theming!',
                                  'subjects': ('design', 'diazo', 'xslt')},
                                 {'title': 'Magic templates in Plone 5',
                                  'subjects': ('templates', 'TAL'),
                                  'url': 'http://www.starzel.de/blog/magic-templates-in-plone-5'}
                                ]">
    <tr>
        <th>Title</th>
        <th>Topics</th>
    </tr>
    <tr tal:repeat="talk talks">
        <td tal:content="talk/title">A talk</td>
        <td tal:define="subjects talk/subjects">
            <span tal:replace="python:', '.join(subjects)">
            </span>
        </td>
    </tr>
</table>
```

2. Turn the title in a link to the URL of the talk if there is one.

Solution

```
<table tal:define="talks python:[{'title': 'Dexterity is the new default!',
                                   'subjects': ('content-types', 'dexterity')},
                                  {'title': 'Mosaic will be the next big thing.',
                                   'subjects': ('layout', 'deco', 'views'),
                                   'url': 'https://www.youtube.com/watch?v=QSNufxaYb1M
↪'},
                                  {'title': 'The State of Plone',
                                   'subjects': ('keynote',) },
                                  {'title': 'Diazo is a powerful tool for theming!',
                                   'subjects': ('design', 'diazo', 'xslt')},
                                  {'title': 'Magic templates in Plone 5',
                                   'subjects': ('templates', 'TAL'),
                                   'url': 'http://www.starzel.de/blog/magic-templates-
↪in-plone-5'}
                                 ]">
    <tr>
        <th>Title</th>
        <th>Topics</th>
    </tr>
    <tr tal:repeat="talk talks">
        <td>
            <a tal:attributes="href talk/url | nothing"
               tal:content="talk/title">
               A talk
            </a>
        </td>
        <td tal:define="subjects talk/subjects">
            <span tal:replace="python:', '.join(subjects)">
            </span>
        </td>
    </tr>
</table>
```

3. If there is no URL, turn it into a link to a google search for that talk's title:

Solution

```
<table tal:define="talks python:[{'title': 'Dexterity is the new default!',
                                   'subjects': ('content-types', 'dexterity')},
                                  {'title': 'Mosaic will be the next big thing.',
                                   'subjects': ('layout', 'deco', 'views'),
                                   'url': 'https://www.youtube.com/watch?v=QSNufxaYb1M
↪'},
                                  {'title': 'The State of Plone',
                                   'subjects': ('keynote',) },
                                  {'title': 'Diazo is a powerful tool for theming!',
                                   'subjects': ('design', 'diazo', 'xslt')},
                                  {'title': 'Magic templates in Plone 5',
                                   'subjects': ('templates', 'TAL'),
                                   'url': 'http://www.starzel.de/blog/magic-templates-
↪in-plone-5'}
                                 ]">
```

```
    <tr>
        <th>Title</th>
        <th>Topics</th>
    </tr>
    <tr tal:repeat="talk talks">
        <td>
            <a tal:define="google_url string:https://www.google.com/search?q=${talk/title}"
               tal:attributes="href talk/url | google_url"
               tal:content="talk/title">
               A talk
            </a>
        </td>
        <td tal:define="subjects talk/subjects">
            <span tal:replace="python:', '.join(subjects)">
            </span>
        </td>
    </tr>
</table>
```

4. Add alternating the CSS classes ‘odd’ and ‘even’ to the <tr>. (`repeat.`*`<name of item in loop>`*`.odd` is True if the ordinal index of the current iteration is an odd number)

   Use some CSS to test your solution:

   ```
   <style type="text/css">
     tr.odd {background-color: #ddd;}
   </style>
   ```

Solution

```
<table tal:define="talks python:[{'title': 'Dexterity is the new default!',
                                  'subjects': ('content-types', 'dexterity')},
                                 {'title': 'Mosaic will be the next big thing.',
                                  'subjects': ('layout', 'deco', 'views'),
                                  'url': 'https://www.youtube.com/watch?v=QSNufxaYb1M'},
                                 {'title': 'The State of Plone',
                                  'subjects': ('keynote',) },
                                 {'title': 'Diazo is a powerful tool for theming!',
                                  'subjects': ('design', 'diazo', 'xslt')},
                                 {'title': 'Magic templates in Plone 5',
                                  'subjects': ('templates', 'TAL'),
                                  'url': 'http://www.starzel.de/blog/magic-templates-in-plone-5'}
                                ]">
    <tr>
        <th>Title</th>
        <th>Topics</th>
    </tr>
    <tr tal:repeat="talk talks"
        tal:attributes="class python: 'odd' if repeat.talk.odd else 'even'">
        <td>
            <a tal:define="google_url string:https://www.google.com/search?q=${talk/title};
                           "
```

```
                tal:attributes="href talk/url | google_url;
                               "
                tal:content="talk/title">
                A talk
            </a>
        </td>
        <td tal:define="subjects talk/subjects">
            <span tal:replace="python:', '.join(subjects)">
            </span>
        </td>
    </tr>
</table>
```

5. Only use python expressions.

Solution

```
<table tal:define="talks python:[{'title': 'Dexterity is the new default!',
                                  'subjects': ('content-types', 'dexterity')},
                                 {'title': 'Mosaic will be the next big thing.',
                                  'subjects': ('layout', 'deco', 'views'),
                                  'url': 'https://www.youtube.com/watch?v=QSNufxaYb1M
↪'},
                                 {'title': 'The State of Plone',
                                  'subjects': ('keynote',) },
                                 {'title': 'Diazo is a powerful tool for theming!',
                                  'subjects': ('design', 'diazo', 'xslt')},
                                 {'title': 'Magic templates in Plone 5',
                                  'subjects': ('templates', 'TAL'),
                                  'url': 'http://www.starzel.de/blog/magic-templates-
↪in-plone-5'}
                                ]">
    <tr>
        <th>Title</th>
        <th>Topics</th>
    </tr>
    <tr tal:repeat="talk python:talks"
        tal:attributes="class python: 'odd' if repeat.talk.odd else 'even'">
        <td>
            <a tal:attributes="href python:talk.get('url', 'https://www.google.com/
↪search?q=%s' % talk['title'])"
                tal:content="python:talk['title']">
                A talk
            </a>
        </td>
        <td tal:content="python:', '.join(talk['subjects'])">
        </td>
    </tr>
</table>
```

6. Use the new syntax of Plone 5

Solution

```
<table tal:define="talks python:[{'title': 'Dexterity is the new default!',
                                  'subjects': ('content-types', 'dexterity')},
                                 {'title': 'Mosaic will be the next big thing.',
                                  'subjects': ('layout', 'deco', 'views'),
                                  'url': 'https://www.youtube.com/watch?v=QSNufxaYb1M
↪'},
                                 {'title': 'The State of Plone',
                                  'subjects': ('keynote',) },
                                 {'title': 'Diazo is a powerful tool for theming!',
                                  'subjects': ('design', 'diazo', 'xslt')},
                                 {'title': 'Magic templates in Plone 5',
                                  'subjects': ('templates', 'TAL'),
                                  'url': 'http://www.starzel.de/blog/magic-templates-
↪in-plone-5'}
                                ]">
    <tr>
        <th>Title</th>
        <th>Topics</th>
    </tr>

    <tr tal:repeat="talk python:talks"
        class="${python: 'odd' if repeat.talk.odd else 'even'}">
        <td>
            <a href="${python:talk.get('url', 'https://www.google.com/search?q=%s' %␣
↪talk['title'])}">
                ${python:talk['title']}
            </a>
        </td>
        <td>
            ${python:', '.join(talk['subjects'])}
        </td>
    </tr>
</table>
```

7. Sort the talks alphabetically by title

Solution

```
<table tal:define="talks python:[{'title': 'Dexterity is the new default!',
                                  'subjects': ('content-types', 'dexterity')},
                                 {'title': 'Mosaic will be the next big thing.',
                                  'subjects': ('layout', 'deco', 'views'),
                                  'url': 'https://www.youtube.com/watch?v=QSNufxaYb1M
↪'},
                                 {'title': 'The State of Plone',
                                  'subjects': ('keynote',) },
                                 {'title': 'Diazo is a powerful tool for theming!',
                                  'subjects': ('design', 'diazo', 'xslt')},
                                 {'title': 'Magic templates in Plone 5',
                                  'subjects': ('templates', 'TAL'),
                                  'url': 'http://www.starzel.de/blog/magic-templates-
↪in-plone-5'}
                                ]">
    <tr>
        <th>Title</th>
        <th>Topics</th>
    </tr>

<?python from operator import itemgetter ?>

    <tr tal:repeat="talk python:sorted(talks, key=itemgetter('title'))"
        class="${python: 'odd' if repeat.talk.odd else 'even'}">
        <td>
            <a href="${python:talk.get('url', 'https://www.google.com/search?q=%s' %␣
↪talk['title'])}">
                ${python:talk['title']}
            </a>
        </td>
        <td>
            ${python:', '.join(talk['subjects'])}
        </td>
    </tr>
</table>
```

METAL and macros

Why is our output so ugly? How do we get our html to render in Plone the UI?

We use METAL (Macro Extension to TAL) to define slots that we can fill and macros that we can reuse.

We add to the `<html>` tag:

```
metal:use-macro="context/main_template/macros/master"
```

And then wrap the code we want to put in the content area of Plone in:

```
<metal:content-core fill-slot="content-core">
    ...
</metal:content-core>
```

This will put our code in a section defined in the main_template called “content-core”.

The template should now look like below when we exlude the last exercise.

Here also added the css-class *listing* to the table. It is one of many css-classes used by Plone that you can reuse in your projects:

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
      lang="en"
      metal:use-macro="context/main_template/macros/master"
      i18n:domain="ploneconf.site">
<body>

<metal:content-core fill-slot="content-core">

<table class="listing"
       tal:define="talks python:[{'title': 'Dexterity is the new default!',
                                   'subjects': ('content-types', 'dexterity')},
                                  {'title': 'Mosaic will be the next big thing.',
                                   'subjects': ('layout', 'deco', 'views'),
                                   'url': 'https://www.youtube.com/watch?v=QSNufxaYb1M
↪'},
                                  {'title': 'The State of Plone',
                                   'subjects': ('keynote',) },
                                  {'title': 'Diazo is a powerful tool for theming!',
                                   'subjects': ('design', 'diazo', 'xslt')},
                                  {'title': 'Magic templates in Plone 5',
                                   'subjects': ('templates', 'TAL'),
                                   'url': 'http://www.starzel.de/blog/magic-templates-in-plone-5'},
                                 ]">
     <tr>
         <th>Title</th>
         <th>Topics</th>
     </tr>

     <tr tal:repeat="talk python:talks"
         class="${python: 'odd' if repeat.talk.odd else 'even'}">
         <td>
             <a href="${python:talk.get('url', 'https://www.google.com/search?q=%s' % talk['title'])}">
                 ${python:talk['title']}
             </a>
         </td>
         <td>
             ${python:', '.join(talk['subjects'])}
         </td>
     </tr>
</table>

</metal:content-core>

</body>
</html>
```

macros in browser views

Define a macro in a new file `macros.pt`

```
<div metal:define-macro="my_macro">
    <p>I can be reused</p>
</div>
```

Register it as a simple BrowserView in zcml:

```
<browser:page
  for="*"
  name="abunchofmacros"
  template="templates/macros.pt"
  permission="zope2.View"
  />
```

Reuse the macro in the template `training.pt`:

```
<div metal:use-macro="context/@@abunchofmacros/my_macro">
    Instead of this the content of the macro will appear...
</div>
```

Which is the same as:

```
<div metal:use-macro="python:context.restrictedTraverse('abunchofmacros')['my_macro']">
    Instead of this the content of the macro will appear...
</div>
```

Restart your Plone instance from the command line, and then open http://localhost:8080/Plone/@@training to see this macro being used in our @@training browser view template.

Accessing Plone from the template

In our template we have access to the context object on which the view is called on, the browser view itself (i.e. all python methods we'll put in the view later on), the request and response objects and with these we can get almost anything.

In templates we can also access other browser views. Some of those exist to provide easy access to methods we often need:

```
tal:define="context_state context/@@plone_context_state;
            portal_state context/@@plone_portal_state;
            plone_tools context/@@plone_tools;
            plone_view context/@@plone;"
```

`@@plone_context_state` The BrowserView `plone.app.layout.globals.context.ContextState` holds useful methods having to do with the current context object such as `is_default_page()`

`@@plone_portal_state` The BrowserView `plone.app.layout.globals.portal.PortalState` holds methods for the portal like `portal_url()`

`@@plone_tools` The BrowserView `plone.app.layout.globals.tools.Tools` gives access to the most important tools like `plone_tools/catalog`

These are very widely used and there are many more.

What we missed

There are some things we did not cover so far:

`tal:condition="exists:expression"` checks if an object or an attribute exists (seldom used)

`tal:condition="nocall:context"` to explicitly not call a callable.

If we refer to content objects, without using the nocall: modifier these objects are unnecessarily rendered in memory as the expression is evaluated.

`i18n:translate` and `i18n:domain` the strings we put in templates can be translated automatically.

There is a lot more about TAL, TALES and METAL that we have not covered. You'll only learn it if you keep reading, writing and customizing templates.

See also:

- http://docs.plone.org/adapt-and-extend/theming/templates_css/template_basics.html
- Using Zope Page Templates: http://docs.zope.org/zope2/zope2book/ZPT.html
- Zope Page Templates Reference: http://docs.zope.org/zope2/zope2book/AppendixC.html
- https://chameleon.readthedocs.io/en/latest/

Customizing Existing Templates

Get the code!

Get the code for this chapter (More info):

```
git checkout zpt_2
```

In this part you will:

- Customize existing templates

Topics covered:

- omelette/packages
- z3c.jbot
- date-formatting and the moment pattern
- listings
- skins

To dive deeper into real Plone data we now look at some existing templates and customize them.

The view for News Items

We want to show the date a News Item is published. This way people can see at a glance if they are looking at current or old news.

To do this we will customize the template that is used to render News Items.

We use `z3c.jbot` for overriding templates. The package already has the necessary configuration in `browser/configure.zcml`.

Find the file `newsitem.pt` in `packages/plone/app/contenttypes/browser/templates/` (in vagrant this directory is in `/home/vagrant/packages`, otherwise it is in your buildout directory).

The file looks like this:

```
<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="context/main_template/macros/master"
    i18n:domain="plone">
<body>

<metal:content-core fill-slot="content-core">
<metal:content-core define-macro="content-core"
                    tal:define="toc context/table_of_contents|nothing;">
  <div id="parent-fieldname-text"
      tal:condition="context/text"
      tal:content="structure python:context.text.output_relative_to(view.context)"
      tal:attributes="class python: toc and 'pat-autotoc' or ''" />
</metal:content-core>
</metal:content-core>
```

```

</body>
</html>
```

Note the following:

- Like almost all Plone templates, it uses *metal:use-macro="context/main_template/macros/master"* to use the main_template
- This template fills the same slot *content-core* as the template you created in the last chapter. This means the heading and description are displayed by the *main_template*.
- The image and image caption that is provided by the behavior is not part of the template.

Copy that file into the folder `browser/overrides/` of our package. If you use vagrant you'd have to use:

```
cp /home/vagrant/packages/plone/app/contenttypes/browser/templates/newsitem.pt /
↪vagrant/buildout/src/ploneconf.site/src/ploneconf/site/browser/overrides/
```

- Rename the new file from `newsitem.pt` to `plone.app.contenttypes.browser.templates.newsitem.pt`. `z3c.jbot` allows you to override templates by putting a file inside a special directory with a *canonical name* (i.e. the path of the file separated by . plus the original filename).
- Restart Plone

Now Plone will use the new file to override the original one.

Edit the new file `plone.app.contenttypes.browser.templates.newsitem.pt` and insert the following before the `<div id="parent-fieldname-text"`...:

```
<p tal:content="python: context.Date()">
    The current Date
</p>
```

Since we use Plone 5 and Chameleon we could also write:

```
<p>
    ${python: context.Date()}
</p>
```

- Open an existing news item in the browser

This will show something like: `2015-02-21T12:01:31+01:00`. Not very user-friendly. Let's extend the code and use one of many helpers Plone offers.

```
<p>
    ${python: plone_view.toLocalizedTime(context.Date())}
</p>
```

This will render `Feb 21,2015`.

- `plone_view` is the BrowserView `Products.CMFPlone.browser.ploneview.Plone` and it is defined in the `main_template` (`Products/CMFPlone/browser/templates/main_template.pt`) of Plone 5 like this `plone_view context/@@plone;` and thus always available.
- The method `toLocalizedTime()` runs a date object through Plone's `translation_service` and returns the Date in the current locales format, thus transforming `2015-02-21T12:01:31+01:00` to `Feb 21,2015`.

The same in a slightly different style:

```
<p tal:define="toLocalizedTime nocall:context/@@plone/toLocalizedTime;
               date python:context.Date()"
   tal:content="python:toLocalizedTime(date)">
        The current Date in its local short format
</p>
```

Here we first get the Plone view and then the method `toLocalizedTime()` and we use `nocall:` to prevent the method `toLocalizedTime()` from being called, since we only want to make it available for later use.

Note: On older Plone versions using Archetypes we used `python:context.toLocalizedTime(context.Date(),longFo` That called the Python script `toLocalizedTime.py` in the Folder `Products/CMFPlone/skins/plone_scripts/`.

That folder `plone_scripts` holds a multitude of useful scripts that are still widely used. But they are all deprecated and most of them are gone in Plone 5 and replaced by proper Python methods in BrowserViews.

We could also leave the formatting to the frontend. Plone 5 comes with the moment pattern that uses the library moment.js to format dates. Try the relative calendar format:

```
<p class="pat-moment"
   data-pat-moment="format:calendar">
    ${python: context.Date()}
</p>
```

Now we should see the date in a user-friendly format like `Today at 12:01 PM`. Experiment with other formats such as `calendar` and `LT`.

The Summary View

We use the view "Summary View" to list news releases. They should also have the date. The template associated with that view is `listing_summary.pt`.

Let's look for the template folder_summary_view.pt:

```
plone/app/contenttypes/browser/templates/listing_summary.pt
```

The file looks like this:

```
<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="context/main_template/macros/master"
    i18n:domain="plone">
<body>

<metal:content-core fill-slot="content-core">
<metal:block use-macro="context/@@listing_view/macros/content-core">

  <metal:entries fill-slot="entries">
    <metal:block use-macro="context/@@listing_view/macros/entries">
      <metal:entry fill-slot="entry">

        <article class="tileItem" tal:define="obj item/getObject">
          <h2 class="tileHeadline" metal:define-macro="listitem">
```

```
            <a class="summary url"
                tal:attributes="href item_link;
                                title item_type"
                tal:content="item_title">
              Item Title
            </a>
          </h2>

          <div metal:use-macro="context/@@listing_view/macros/document_byline"></div>

          <div class="tileImage"
               tal:condition="item_has_image"
               tal:attributes="class python: 'tileImage' if item_description else
↪'tileImageNoFloat'">
            <a tal:attributes="href item_link">
              <img tal:define="scales obj/@@images;
                               scale python:scales.scale('image', 'thumb')"
                  tal:replace="structure python:scale and scale.tag() or None" />
            </a>
          </div>

          <div class="tileBody" tal:condition="item_description">
            <span class="description" tal:content="item_description">
              description
            </span>
          </div>

          <div class="tileFooter">
            <a tal:attributes="href item_link"
                i18n:translate="read_more">
              Read More&hellip;
            </a>
          </div>

          <div class="visualClear"><!-- --></div>

        </article>

      </metal:entry>
    </metal:block>
  </metal:entries>

</metal:block>
</metal:content-core>

</body>
</html>
```

Note the following:

- Unlike `newsitem.pt` the file does not display data from a context but obviously pre-defined variables like *item*, *item_link*, *item_type* or *item_description*.
- It reuses multiple macros of a view *context/@@listing_view*.
- The variables are most likely defined in the macro *entries* of that view.

Copy it to `browser/overrides/` and rename it to `plone.app.contenttypes.browser.templates.listing_summar`

Add the following after line 28:

```
<p tal:condition="python:item_type == 'News Item'">
  ${python:plone_view.toLocalizedTime(item.Date())}
</p>
```

After you restart the instance and look at the new folder again you'll see the dates. `z3c.jbot` needs a restart to pick up the new file. When you only change a existing override you don't have to restart.

The addition renders the date of the respective objects that the template iterates over (hence `item` instead of `context` since `context` would be either a collection aggregating the news items or a folder containing a news item).

The date is only displayed if the variable `item_type` is `News Item`.

Let's take a closer look at that template. How does it know that `item_type` is the name of the content type?

The first step to uncovering that secret is line 14 of `listing_summary.pt`:

```
<metal:block use-macro="context/@@listing_view/macros/entries">
```

`use-macro` tells Plone to reuse the macro `entries` from the view `listing_view`. That view is defined in `packages/plone/app/contenttypes/browser/configure.zcml`. It uses the template `plone/app/contenttypes/browser/templates/listing.pt`. That makes overriding that much easier.

That template `listing.pt` defines the slot `entries` like this:

```
<metal:listingmacro define-macro="listing">
  <tal:results define="batch view/batch">
    <tal:listing condition="batch">
      <div class="entries" metal:define-slot="entries">
        <tal:entries repeat="item batch" metal:define-macro="entries">
          <tal:block tal:define="obj item/getObject;
                                 item_url item/getURL;
                                 item_id item/getId;
                                 item_title item/Title;
                                 item_description item/Description;
                                 item_type item/PortalType;
                                 item_modified item/ModificationDate;
                                 item_created item/CreationDate;
                                 item_icon item/getIcon;
                                 item_type_class python:'contenttype-' + view.
↪normalizeString(item_type);
                                 item_wf_state item/review_state;
                                 item_wf_state_class python:'state-' + view.
↪normalizeString(item_wf_state);
                                 item_creator item/Creator;
                                 item_link python:item_type in view.use_view_action␣
↪and item_url+'/view' or item_url;
                                 item_has_image python:view.has_image(obj);
                                 item_is_event python:view.is_event(obj)">

...
```

Here the `item_type` is defined as `item_type item/PortalType`. Let's dig a little deeper and find out what `item` and `PortalType` are.

`tal:repeat="item batch"` tells the template to iterate over an iterable `batch` which is defined as `batch view/batch`.

`view` is always the BrowserView for which the template is registered. In our case this is either `plone.app.contenttypes.browser.collection.CollectionView` if you called that view on a collection, or plone.app.contenttypes.browser.folder.FolderView for folders. You might remember that both are defined in configure.zcml

Luckily the first is a class that inherits from the second:

```
class CollectionView(FolderView):
```

batch() is a method in FolderView that turns results into batches. results exists in both classes. This means, in case the item we are looking at is a collection, the method results() of CollectionView, will be used; and in case it's a folder, the one in FolderView.

So *batch* is a list of items. The way it is created is actually pretty complicated and makes use of a couple of packages to create a filtered (through plone.app.querystring) list of optimized representations (through plone.app.contentlisting) of items. For now it is enough to know that *item* represents one of the items in the list of News Items.

The template listing_summary.pt is extraordinary in its heavy use of nested macros. Most of the templates you will write are much simpler and easier to read.

It can be hard to understand templates as complicated as these, but there is help to be found if you know Python: use pdb to debug templates line by line.

Add the following to line 29 just before our additions:

```
<?python import pdb; pdb.set_trace() ?>
```

When you reload the page and look at the terminal you see you have the pdb console and can inspect the template at its current state by looking at the variable *econtext*. You can now simply look up what *item ' and 'PortalType* are:

```
(pdb) pp econtext
[...]
'context': <Collection at /Plone/news/aggregator>,
'context_state': <Products.Five.metaclass.ContextState object at 0x10b7f50d0>,
'default': <object object at 0x100294c50>,
'dummy': None,
'here': <Collection at /Plone/news/aggregator>,
'isRTL': False,
'item': <plone.app.contentlisting.catalog.CatalogContentListingObject instance at /
↪Plone/news/hot-news>,
'item_created': '2016-10-08T15:04:17+02:00',
'item_creator': 'admin',
[...]
(pdb) item = econtext['item']
(pdb) item
<plone.app.contentlisting.catalog.CatalogContentListingObject instance at /Plone/news/
↪hot-news>
```

As discovered above, *item* is a instance of plone.app.contentlisting.catalog.CatalogContentListingObject. It has several methods and properties:

```
(pdb) pp dir(item)
[...]
'Language',
'ModificationDate',
'PortalType',
'Publisher',
'ReviewStateClass',
'Rights',
[...]
```

PortalType is a method that returns the name of the items content-type.

```
(pdb) item.PortalType()
'News Item'
```

Note: In Plone 4 without `plone.app.contenttypes` the template to customize would be `folder_summary_view.pt`, a skin template for Archetypes that can be found in the folder `Products/CMFPlone/skins/plone_content/`. The customized template would be `Products.CMFPlone.skins.plone_content.folder_summary_view.pt`.

The Archetypes template for News Items is `newsitems_view.pt` from the same folder. The customized template would then have to be named `Products.CMFPlone.skins.plone_content.newsitems_view.pt`.

Keep in mind that not only the names and locations have changed but also the content and the logic behind them!

Finding the right template

We changed the display of the listing of news items at http://localhost:8080/Plone/news. But how do we know which template to customize?

If you don't know which template is used by the page you're looking at, you can make an educated guess. Start a debug session or use `plone.app.debugtoolbar`.

1. We could check the HTML with Firebug and look for a structure in the content area that looks unique. We could also look for the CSS class of the body

   ```
   <body class="template-summary_view portaltype-collection site-Plone section-news␣
   ↪subsection-aggregator icons-on userrole-anonymous" dir="ltr">
   ```

   The class `template-summary_view` tells us that the name of the view (but not necessarily the name of the template) is `summary_view`. So we could search all `*.zcml`-Files for `name="summary_view"` or search all templates called `summary_view.pt` and probably find the view and also the corresponding template. But only probably because it would not tell us if the template is already being overridden.

   A foolproof way to verify your guess is to modify the template and reload the page. If your modification shows up you obviously found the correct file.

2. The safest method is using `plone.app.debugtoolbar`. We already have it in our buildout and only need to install it. It adds a "Debug" dropdown menu on top of the page. The section "Published" shows the complete path to the template that is used to render the page you are seeing.

3. The debug session to find the template is a little more complicated. Since we have `Products.PDBDebugMode` in our buildout we can call `/pdb` on our page. We cannot put a *pdb* in the templates since we do not know (yet) which template to put the *pdb* in.

   The object that the URL points to is by default `self.context`. But the first problem is that the URL we're seeing is not the URL of the collection we want to modify. This is because the collection is the default page of the folder `news`.

   ```
   (Pdb) self.context
   <Folder at /Plone/news>
   (Pdb) obj = self.context.aggregator
   (Pdb) obj
   <Collection at /Plone/news/aggregator>
   (Pdb) context_state = obj.restrictedTraverse('@@plone_context_state')
   (Pdb) template_id = context_state.view_template_id()
   (Pdb) template_id
   ```

```
'summary_view'
(Pdb) view = obj.restrictedTraverse('summary_view')
(Pdb) view
<Products.Five.metaclass.SimpleViewClass from /Users/philip/.cache/buildout/eggs/
↪plone.app.contenttypes-1.1b2-py2.7.egg/plone/app/contenttypes/browser/templates/
↪summary_view.pt object at 0x10b00cd90>
view.index.filename
u'/Users/philip/workspace/training_without_vagrant/src/ploneconf.site/ploneconf/
↪site/browser/template_overrides/plone.app.contenttypes.browser.templates.
↪summary_view.pt'
```

Now we see that we already customized the template.

Here is a method that could be used in a view or viewlet to display that path:

```
def get_template_path(self):
    context_state = api.content.get_view(
        'plone_context_state', self.context, self.request)
    view_template_id = context_state.view_template_id()
    view = self.context.restrictedTraverse(view_template_id)
    return view.index.filename
```

skin templates

Why don't we always only use templates? Because we might want to do something more complicated than get an attribute from the context and render its value in some html tag.

There is a deprecated technology called 'skin templates' that allows you to simply add some page template (e.g. 'old_style_template.pt') to a certain folder in the ZMI or your egg and you can access it in the browser by opening a url like http://localhost:8080/Plone/old_style_template and it will be rendered. But we don't use it and you too should not, even though these skin templates are still all over Plone.

Since we use `plone.app.contenttypes` we do not encounter many skin templates when dealing with content any more. But more often than not you'll have to customize an old site that still uses skin templates.

Skin templates and Python scripts in `portal_skins` are deprecated because:

- they use restricted Python
- they have no nice way to attach Python code to them
- they are always callable for everything (they can't easily be bound to an interface)

Summary

- Overriding templates with `z3c.jbot` is easy.
- Understanding templates can be hard.
- Use plone.app.debugtoolbar and pdb; they are there to help you.
- Skin templates are deprecated; you will probably only encounter them when you work on Plone 4.

Views II: A Default View for "Talk"

Get the code!

Get the code for this chapter (More info):

```
git checkout views_2
```

In this part you will:

- Register a view with a python class
- Write a template used in the default view for talks

Topics covered:

- View classes
- BrowserView and DefaultView
- displaying data from fields

View Classes

Earlier we wrote a demo view which we also used to experiment with page templates. Now we are going to enhance that view so that it will have some python code, in addition to a template. Let us have a look at the zcml and the code.

browser/configure.zcml

```
<configure xmlns="http://namespaces.zope.org/zope"
    xmlns:browser="http://namespaces.zope.org/browser"
    i18n_domain="ploneconf.site">

    <browser:page
       name="demoview"
       for="*"
       class=".views.DemoView"
       template="templates/training.pt"
       permission="zope2.View"
       />

</configure>
```

We are adding a file called `views.py` in the `browser` folder.

browser/views.py

```
from Products.Five.browser import BrowserView

class DemoView(BrowserView):
    """ This does nothing so far
    """

    def the_title(self):
        return u'A list of great trainings:'
```

In the template `training.pt` we can now use this view as *view* and access all its methods and properties:

```
<h2 tal:content="python: view.the_title()" />
```

The logic contained in that file can now be moved to the class:

```python
# -*- coding: utf-8 -*-
from Products.Five.browser import BrowserView
from operator import itemgetter


class DemoView(BrowserView):
    """A demo listing"""

    def the_title(self):
        return u'A list of talks:'

    def talks(self):
        results = []
        data = [
            {'title': 'Dexterity is the new default!',
             'subjects': ('content-types', 'dexterity')},
            {'title': 'Mosaic will be the next big thing.',
             'subjects': ('layout', 'deco', 'views'),
             'url': 'https://www.youtube.com/watch?v=QSNufxaYb1M'},
            {'title': 'The State of Plone',
             'subjects': ('keynote',)},
            {'title': 'Diazo is a powerful tool for theming!',
             'subjects': ('design', 'diazo', 'xslt')},
            {'title': 'Magic templates in Plone 5',
             'subjects': ('templates', 'TAL'),
             'url': 'http://www.starzel.de/blog/magic-templates-in-plone-5'},
        ]
        for item in data:
            try:
                url = item['url']
            except KeyError:
                url = 'https://www.google.com/search?q=%s' % item['title']
            talk = dict(
                title=item['title'],
                subjects=', '.join(item['subjects']),
                url=url
            )
            results.append(talk)
        return sorted(results, key=itemgetter('title'))
```

And the template will now be much simpler.

```html
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
      lang="en"
      metal:use-macro="context/main_template/macros/master"
      i18n:domain="ploneconf.site">
<body>

<metal:content-core fill-slot="content-core">

<h2 tal:content="python: view.the_title()" />

<table class="listing">
    <tr>
        <th>Title</th>
        <th>Topics</th>
    </tr>

    <tr tal:repeat="talk python:view.talks()">
        <td>
            <a href="${python:talk['url']}">
                ${python:talk['title']}
            </a>
        </td>
        <td>
            ${python:talk['subjects']}
        </td>
    </tr>
</table>

</metal:content-core>

</body>
</html>
```

The default view

Using a view you can now create a nice view for talks in `views.py`. First we will not write any methods for *view* but instead access the fields from the talk-schema as *context.<fieldname>*.

Register a view *talkview* in `browser/configure.zcml`:

```
<browser:page
   name="talkview"
   for="*"
   layer="zope.interface.Interface"
   class=".views.TalkView"
   template="templates/talkview.pt"
   permission="zope2.View"
   />
```

`browser/views.py`

```
class TalkView(BrowserView):
    """ The default view for talks"""
```

Add the template `templates/talkview.pt`:

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
    lang="en"
    metal:use-macro="context/main_template/macros/master"
    i18n:domain="ploneconf.site">
<body>
    <metal:content-core fill-slot="content-core">
        <p>Suitable for <em tal:content="python: ', '.join(context.subject)"></em>
        </p>

        <div tal:condition="python: context.details"
             tal:content="structure python: context.details.output" />

        <div tal:content="python: context.speaker">
            User
        </div>
    </metal:content-core>
```

```
</body>
</html>
```

After a restart, we can test our view by going to a talk and adding */talkview* to the url.

Using helper-methods from `DefaultView`

Dexterity comes with a nice helper-class suited for views of content-types: The `DefaultView` base class in `plone.dexterity`. It only works for Dexterity Objects and has some very useful properties 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 widget list to get widgets in that fieldset

You can now change the `TalkView` to use that

```python
from plone.dexterity.browser.view import DefaultView

...

class TalkView(DefaultView):
    """ The default view for talks
    """
```

The template `templates/talkview.pt` still works but now you can modify it to use the pattern `view/w/<fieldname>/render` to render the widgets:

```html
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
    lang="en"
    metal:use-macro="context/main_template/macros/master"
    i18n:domain="ploneconf.site">
<body>
    <metal:content-core fill-slot="content-core">
        <p>Suitable for <em tal:replace="structure view/w/audience/render"></em>
        </p>

        <div tal:content="structure view/w/details/render" />

        <div tal:content="context/speaker">
            User
        </div>
    </metal:content-core>
</body>
</html>
```

After a restart, we can test the modified view by going to a talk and adding */talkview* to the url.

We should tell Plone that the talkview should be used as the default view for talks instead of the built-in view.

This is a configuration that you can change during runtime and is stored in the database, as such it is also managed by GenericSetup profiles.

open `profiles/default/types/talk.xml`:

```xml
...
<property name="default_view">talkview</property>
<property name="view_methods">
    <element value="talkview"/>
    <element value="view"/>
</property>
...
```

We will have to either reinstall our add-on or run the GenericSetup import step `typeinfo` so Plone learns about the change.

Note: To change it ttw got to the ZMI (http://localhost:8080/Plone/manage), go to `portal_types` and select the type for which the new view should be selectable (*talk*). Now add `talkview` to the list *Available view methods*. Now the new view is available in the menu *Display*. To make it the default view enter it in `Default view method`.

Let's improve the talkview to show all the info we want.

`templates/talkview.pt`:

```html
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en"
      metal:use-macro="context/main_template/macros/master"
      i18n:domain="ploneconf.site">
<body>
    <metal:content-core fill-slot="content-core">

        <p>
            <span tal:content="context/type_of_talk">
                Talk
            </span>
            suitable for
            <span tal:replace="structure view/w/audience/render">
                Audience
            </span>
        </p>

        <div tal:content="structure view/w/details/render">
            Details
        </div>

        <div class="newsImageContainer">
            <img tal:condition="python:getattr(context, 'image', None)"
                 tal:attributes="src string:${context/absolute_url}/@@images/image/thumb" />
        </div>

        <div>
            <a class="email-link" tal:attributes="href string:mailto:${context/email}">
                <strong tal:content="context/speaker">
                    Jane Doe
                </strong>
            </a>
            <div tal:content="structure view/w/speaker_biography/render">
                Biography
            </div>
        </div>

    </metal:content-core>
</body>
</html>
```

Exercise

Add the new choice field “room” to the Talk type (TTW) and display it below Audience in the browser view, it should contain the following data:

- Title: Room
- Possible values: Room 101, Room 102, Auditorium

Solution

- Go to http://localhost:8080/Plone/dexterity-types/talk/@@fields and add the new field
- Add the new HTML below the audience part:

```
<p>
    <span tal:replace="structure view/w/room/render">
        Room
    </span>
</p>
```

Behind the scenes

```
from Products.Five.browser import BrowserView

class DemoView(BrowserView):

    def __init__(self, context, request):
        self.context = context
        self.request = request

    def __call__(self):
        # Implement your own actions

        # This renders the template that was registered in zcml like this:
        #   template="templates/training.pt"
        return super(DemoView, self).__call__()
        # If you don't register a template in zcml the Superclass of
        # DemoView will have no __call__-method!
        # In that case you have to call the template like this:
        # from Products.Five.browser.pagetemplatefile import ViewPageTemplateFile
        # class DemoView(BrowserView):
        # template = ViewPageTemplateFile('templates/training.pt')
        # def __call__(self):
        #    return self.template()
```

Do you remember the term `MultiAdapter`? The browser page is just a MultiAdapter. The ZCML statement `browser:page` registers a `MultiAdapter` and adds additional things needed for a browser view.

An adapter adapts things, a `MultiAdapter` adapts multiple things.

When you enter a url, Zope tries to find an object for it. At the end, when Zope does not find any more objects but there is still a path item left, or there are no more path items, Zope looks for an adapter that will reply to the request.

The adapter adapts the request and the object that Zope found with the URL. The adapter class gets instantiated with the objects to be adapted, then it gets called.

The code above does the same thing that the standard implementation would do. It makes `context` and `request` available as variables on the object.

I have written down these methods because it is important to understand some important concepts.

The `__init__()` method gets called while Zope is still *trying* to find a view. At that phase, the security has not been resolved. Your code is not security checked. For historical reasons, many errors that happen in the `__init__()` method can result in a page not found error instead of an exception.

Use the `__init__()` method to do as little as possible, if at all. Instead, you have the guarantee that the `__call__()` method is called before anything else (but after the `__init__()` method). It has the security checks in place and so on.

From a practical standpoint, consider the `__call__()` method your `__init__()` method, the biggest difference is that this method is supposed to return the HTML already. Let your base class handle the HTML generation.

See also:

http://docs.plone.org/develop/plone/views/browserviews.html

Views III: A Talk List

Get the code!

Get the code for this chapter (More info):

```
git checkout views_3
```

In this part you will:

- Write a python class to get all talks from the catalog
- Write a template to display the talks
- Improve the table

Topics covered:

- BrowserView
- plone.api
- portal_catalog
- brains and objects
- tables

Now we don't want to provide information about one specific item but on several items. What now? We can't look at several items at the same time as context.

Using portal_catalog

Let's say we want to show a list of all the talks that were submitted for our conference. We can just go to the folder and select a display method that suits us. But none does because we want to show the target audience in our listing.

So we need to get all the talks. For this we use the python class of the view to query the catalog for the talks.

The catalog is like a search engine for the content on our site. It holds information about all the objects as well as some of their attributes like title, description, workflow_state, keywords that they were tagged with, author, content_type, its path in the site etc. But it does not hold the content of "heavy" fields like images or files, richtext fields and fields that we just defined ourselves.

It is the fast way to get content that exists in the site and do something with it. From the results of the catalog we can get the objects themselves but often we don't need them, but only the properties that the results already have.

browser/configure.zcml

```
<browser:page
   name="talklistview"
   for="*"
   layer="zope.interface.Interface"
   class=".views.TalkListView"
   template="templates/talklistview.pt"
   permission="zope2.View"
   />
```

browser/views.py

```
from Products.Five.browser import BrowserView
from plone import api
from plone.dexterity.browser.view import DefaultView

[...]

class TalkListView(BrowserView):
    """ A list of talks
    """

    def talks(self):
        results = []
        brains = api.content.find(context=self.context, portal_type='talk')
        for brain in brains:
            talk = brain.getObject()
            results.append({
                'title': brain.Title,
                'description': brain.Description,
                'url': brain.getURL(),
                'audience': ', '.join(talk.audience),
                'type_of_talk': talk.type_of_talk,
                'speaker': talk.speaker,
                'uuid': brain.UID,
                })
        return results
```

We query the catalog with two parameters. The catalog returns only items for which **both** apply:

- `context=self.context`
- `portal_type='talk'`

We pass a object as *context* to query only for content in the current path. Otherwise we'd get all talks in the whole site. If we moved some talks to a different part of the site (e.g. a sub-conference for universities with a special talk list) we might not want so see them in our listing. We also query for the *portal_type* so we only find talks.

Note: We use the method `find()` in `plone.api` to query the catalog. It is one of many convenience-methods provided as a wrapper around otherwise more complex api's. If you query the catalog direcly you'd have to first get the catalog, and pass it the path for which you want to find items:

```
portal_catalog = api.portal.get_tool('portal_catalog')
current_path = '/'.join(self.context.getPhysicalPath())
brains = portal_catalog(path=current_path, portal_type='talk')
```

We iterate over the list of results that the catalog returns us.

We create a dictionary that holds all the information we want to show in the template. This way we don't have to put any complex logic into the template.

brains and objects

Objects are normally not loaded into memory but lie dormant in the ZODB Database. Waking objects up can be slow, especially if you're waking up a lot of objects. Fortunately our talks are not especially heavy since they are:

- dexterity-objects which are lighter than their archetypes brothers
- relatively few since we don't have thousands of talks at our conference

We want to show the target audience but that attribute of the talk content type is not in the catalog. This is why we need to get to the objects themselves.

We could also add a new index to the catalog that will add 'audience' to the properties of brains, but we should weigh the pros and cons:

- talks are important and thus most likely always in memory
- prevent bloating of catalog with indexes

Note: The code to add such an index would look like this:

```
from plone.indexer.decorator import indexer
from ploneconf.site.talk import ITalk

@indexer(ITalk)
def talk_audience(object, **kw):
     return object.audience
```

We'd have to register this factory function as a named adapter in the `configure.zcml`. Assuming you've put the code above into a file named `indexers.py`

```
<adapter name="audience" factory=".indexers.talk_audience" />
```

We will add some indexers later on.

Why use the catalog at all? It checks for permissions, and only returns the talks that the current user may see. They might be private or hidden to you since they are part of a top secret conference for core developers (there is no such thing!).

Most objects in Plone act like dictionaries, so you can do `context.values()` to get all its contents.

For historical reasons some attributes of brains and objects are written differently.

```
>>> obj = brain.getObject()

>>> obj.title
u'Talk-submission is open!'

>>> brain.Title == obj.title
True

>>> brain.title == obj.title
False
```

Who can guess what `brain.title` will return since the brain has no such attribute?

Note: Answer: Acquisition will get the attribute from the nearest parent. `brain.__parent__` is `<CatalogTool at /Plone/portal_catalog>`. The attribute `title` of the `portal_catalog` is 'Indexes all content in the site'.

Acquisition can be harmful. Brains have no attribute 'getLayout' `brain.getLayout()`:

```
>>> brain.getLayout()
'folder_listing'

>>> obj.getLayout()
'newsitem_view'

>>> brain.getLayout
<bound method PloneSite.getLayout of <PloneSite at /Plone>>
```

The same is true for methods:

```
>>> obj.absolute_url()
'http://localhost:8080/Plone/news/talk-submission-is-open'
>>> brain.getURL() == obj.absolute_url()
True
>>> brain.getPath() == '/'.join(obj.getPhysicalPath())
True
```

Querying the catalog

The are many catalog indexes to query. Here are some examples:

```
>>> portal_catalog = getToolByName(self.context, 'portal_catalog')
>>> portal_catalog(Subject=('cats', 'dogs'))
[]
>>> portal_catalog(review_state='pending')
[]
```

Calling the catalog without parameters returns the whole site:

```
>>> portal_catalog()
[<Products.ZCatalog.Catalog.mybrains object at 0x1085a11f0>, <Products.ZCatalog.
↪Catalog.mybrains object at 0x1085a12c0>, <Products.ZCatalog.Catalog.mybrains object␣
↪at 0x1085a1328>, <Products.ZCatalog.Catalog.mybrains object at 0x1085a13 ...
```

See also:

http://docs.plone.org/develop/plone/searching_and_indexing/query.html

Exercises

Since you now know how to query the catalog it is time for some exercise.

Exercise 1

Add a method `get_news()` to `TalkListView` that returns a list of brains of all News Items that are published and sort them in the order of their publishing-date.

Solution

```
def get_news(self):

    portal_catalog = api.portal.get_tool('portal_catalog')
    return portal_catalog(
        portal_type='News Item',
        review_state='published',
        sort_on='effective',
    )
```

Exercise 2

Add a method that returns all published keynotes as objects.

Solution

```
def keynotes(self):

    portal_catalog = api.portal.get_tool('portal_catalog')
    brains = portal_catalog(
        portal_type='Talk',
        review_state='published')
    results = []
    for brain in brains:
        # There is no catalog-index for type_of_talk so we must check
        # the objects themselves.
        talk = brain.getObject()
        if talk.type_of_talk == 'Keynote':
            results.append(talk)
    return results
```

The template for the listing

Next you create a template in which you use the results of the method 'talks'.

Try to keep logic mostly in python. This is for two reasons:

Readability: It's much easier to read python than complex tal-structures

Speed: Python-code is faster than code executed in templates. It's also easy to add caching to methods.

DRY: In Python you can reuse methods and easily refactor code. Refactoring TAL usually means having to do big changes in the html-structure which results in incomprehensible diffs.

The MVC-Schema does not directly apply to Plone but look at it like this:

Model: the object

View: the template

Controller: the view

The view and the controller are very much mixed in Plone. Especially when you look at some of the older code of Plone you'll see that the policy of keeping logic in python and representation in templates was not always enforced.

But you should nevertheless do it! You'll end up with more than enough logic in the templates anyway.

Add this simple table to `templates/talklistview.pt`:

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en"
      metal:use-macro="context/main_template/macros/master"
      i18n:domain="ploneconf.site">
<body>
  <metal:content-core fill-slot="content-core">
  <table class="listing"
         id="talks"
         tal:define="talks python:view.talks()">
    <thead>
      <tr>
        <th>Title</th>
        <th>Speaker</th>
        <th>Audience</th>
      </tr>
    </thead>
    <tbody>
      <tr tal:repeat="talk talks">
        <td>
          <a href=""
             tal:attributes="href python:talk['url'];
                             title python:talk['description']"
             tal:content="python:talk['title']">
             The 7 sins of plone-development
          </a>
        </td>
        <td tal:content="python:talk['speaker']">
            Philip Bauer
        </td>
        <td tal:content="python:talk['audience']">
            Advanced
        </td>
      </tr>
      <tr tal:condition="not:talks">
        <td colspan=3>
            No talks so far :-(
        </td>
      </tr>
    </tbody>
  </table>

  </metal:content-core>
</body>
</html>
```

Again we use `class="listing"` to give the table a nice style.

There are some some things that need explanation:

`tal:define="talks python:view.talks()"` This defines the variable *talks*. We do thins since we reuse it later and don't want to call the same method twice. Since TAL's path expressions for the lookup of values in dictionaries is the same as for the attributes of objects and methods of classes we can write `view/talks` as we could `view/someattribute`. Handy but sometimes irritating since from looking at the page template alone we often have no way of knowing if something is an attribute, a method or the value of a dict.

`tal:repeat="talk talks"` This iterates over the list of dictionaries returned by the view. Each `talk` is one of the dictionaries that are returned by this method.

`tal:content="python:talk['speaker']"` 'speaker' is a key in the dict 'talk'. We could also write `tal:content="talk/speaker"`

`tal:condition="not:talks"` This is a fallback if no talks are returned. It then returns an empty list (remember `results = []`?)

Note: We could also write `python:not talks` like we could also write `tal:repeat="talk python:talks"` for the iteration. For simple cases as these path-statements are sometimes fine. On the other hand: If `talks` would be a callable we woul need to use `nocall:talks`, so maybe it would be better to always use `python:`.

Exercise

Modify the view to only use path-expressions. This is **not** best-practice but there is plenty of code in Plone and in Addons so you have to know how to use them.

Solution

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en"
      metal:use-macro="context/main_template/macros/master"
      i18n:domain="ploneconf.site">
<body>
  <metal:content-core fill-slot="content-core">
  <table class="listing" id="talks"
         tal:define="talks view/talks">
    <thead>
      <tr>
        <th>Title</th>
        <th>Speaker</th>
        <th>Audience</th>
      </tr>
    </thead>
    <tbody>
      <tr tal:repeat="talk talks">
        <td>
          <a href=""
```

```
             tal:attributes="href talk/url;
                             title talk/description"
             tal:content="talk/title">
             The 7 sins of plone-development
          </a>
        </td>
        <td tal:content="talk/speaker">
            Philip Bauer
        </td>
        <td tal:content="talk/audience">
            Advanced
        </td>
      </tr>
      <tr tal:condition="not:talks">
        <td colspan=3>
            No talks so far :-(
        </td>
      </tr>
    </tbody>
  </table>

  </metal:content-core>
</body>
</html>
```

Setting a custom view as default view on an object

We don't want to always have to append `/@@talklistview` to our folder to get the view. There is a very easy way to set the view to the folder using the ZMI.

If we append `/manage_propertiesForm` we can set the property "layout" to `talklistview`.

To make views configurable so that editors can choose them we have to register the view for the content type at hand in its FTI. To enable it for all folders we add a new file `profiles/default/types/Folder.xml`

```xml
<?xml version="1.0"?>
<object name="Folder">
 <property name="view_methods" purge="False">
  <element value="talklistview"/>
 </property>
</object>
```

After re-applying the typeinfo profile of our add-on (or simply reinstalling it) the content type "Folder" is extended with our additional view method and appears in the display dropdown.

The `purge="False"` appends the view to the already existing ones instead of replacing them.

Summary

We created a nice listing, that can be called at any place in the website.

Testing in Plone

Get the code!

Get the code for this chapter (More info):

```
git checkout testing
```

In this chapter we:

- Write tests

Topics covered:

- Testing best practices
- Internals of Plone

Types of tests

Plone is using some common terminology for types of tests you might have heard elsewhere. But in Plone, these terms are usually used to differentiate the technical difference between the types of test.

Unit tests

These match the normal meaning the most. Unit tests test a unit in isolation. That means there is no database, no component architecture and no browser. This means the code is very fast and it can mean that you can't test all that much if your code mostly interacts with other components.

A unit test for a browser view would create an instance of the view directly. That means it is your responsibility to provide a proper context and a proper request. You can't really test user-dependent behavior because you just mock a Request object imitating a user or not. This code might be broken with the next version of Plone without the test failing.

On the other hand, testing a complex rule with many different outcomes is still best tested in a unit test, because they are very fast.

Integration tests

Integration tests in Plone mean you have a real database and your component architecture. You can identify an integration test by the layer it is using which is based on a layer with integration in its name. We will explain shortly what a layer is.

Integration tests also means your test is still quite fast, because the transaction mechanisms are used for test isolation. What does that mean? After each test, the transaction gets canceled and you have the database in the same state as before. It still takes a while to set up the test layer, but running each test is quite fast. But this also means you cannot commit a transaction. Most code does not commit transactions and this is not an issue.

Functional tests

Functional tests in Plone have a real database and a component architecture, like Integration tests. In addition, you can simulate a browser in python code. When this browser tries to access a page, the complete transaction machinery is in use. For this to work, the test layer wraps the database into a demostorage. A Demostorage is for demonstration. A demostorage wraps a regular storage. When something gets written into the database, the demostorage stores it into memory or temporary fields. On reading it either returns what has been saved in memory or what is in the underlaying storage. After each test, the demostorage is wiped. This should make it nearly as fast as integration tests, but there is an additional overhead, when requests get through the transaction machinery. Also, the browser is pure python code. It knows nothing about javascript. You cannot test your javascript code with functional tests

Acceptance tests

Acceptance tests are usually tests that can assert that an application would pass the requirements the customer gave. This implies that acceptance tests test the complete functionality and that they either allow the customer to understand what is being tested or at least clearly map to business requirements. In Plone, acceptance tests are tests written with the so called robot framework. Here you write tests in something resembling a natural language and which is driven by a real web browser. This implies you can also test Javascript. This is the slowest form of testing but also the most complete. Also, acceptance tests aren't limited to the original form of acceptance tests, but also for normal integration tests.

Javascript tests

So far, it looks like we only have acceptance tests for testing javascript. Acceptance tests are also very new. This means we had no test story for testing javascript. In Plone 5, we have the mockup framework to write javascript components and the mockup framework provides also scaffolding for testing Javascript with xxx. While these tests use a real browser of some sort, they fall into the category of unit tests, because you have no database Server available to generate proper html.

Doctests

Doctests are a popular way to write tests in documentation. Doctests parse documentation for code that has special formatting and runs the code and compares it with the output suggested in the documentation. Doctests are hard to debug, because there is no easy way to use a debugger in doctests. Doctests have a bad reputation, because when it came around, people thought they could write documentation and tests in one go. This resulted in packages like zope.component, where the documentation on pypi slowly transforms into half sentences split up by 5-10 lines of code testing an obscure feature that the half sentence does not properly explain. In Plone, this form of testing is not very common. We would like to transform our documentation to be testable with doctests.

Writing tests

Writing tests is an art. If your testsuite needs half an hour to run, it loses a lot of value. If you limit yourself to unit tests and fake everything, you miss many bugs, either because Plone works differently than what you thought, or the next Plone versions run differently from today's. On the other hand, integration tests are not only slower, but often create test failures far away from the actual error in the code. Not only do the tests run more slowly, it also takes longer to debug why they fail. Here are some good rules to take into account.

If you need to write many test cases for a browser view, you might want to factor this out into a component of its own, in such a way that this component can easily be tested with unit tests. If, for example, you have a list view that shall do a specific way of sorting, depending on gender, language and browser of a user, write a component that takes a list of names to sort, gender, language and browser as strings. This code can easily be tested for all combinations in unit tests, while extracting gender, language and browser from a request object takes only a few functional tests.

Try not to mock code. The mocked code you generate mocks Plone in the version you are using today. The next version might work differently.

Do not be afraid to rewrite your code for better testability. It pays off.

If you have highly complex code, think about structuring code and data structures in such a way that they have no side effects. For one customer I wrote a complex ruleset of about 400 lines of code. A lot of small methods that have no side effects. It took a bit to write that code and corresponding tests, but as of today this code did not have a single failure.

Steal from others. Unfortunately, it sometimes takes an intrinsic knowledge to know how to test some functionality. Some component functionality that is automatically handled by the browser must be done by hand. And the component documentation has been referenced in this chapter as a terrible example already. So, copy your code from somewhere else.

Normally, you write a test that tests one thing only. Don't be afraid to break that rule when necessary. If, for example, you built some complex logic that involves multiple steps, don't shy away from writing a longer test showing the normal, good case. Add lots of comments explaining in each step what is happening, why and how. This helps other developers and the future you.

Plone tests

Plone is a complex system to run tests in. Because of this, we use a functionality from zope.testrunner: layers. We use the well known unittest framework which exhibits the same ideas as nearly every unittest framework out there. In addition for test setups we have the notion of layers. A layer is a test setup that can be shared. This way, you can run tests from 20 different testsuites but not each testsuite sets up their own complete Plone site. Instead, you use a Layer, and the testrunner takes care that every testsuite sharing a layer are run together.

Usually, you create three layers on your own, an integration layer, a functional layer and an acceptance test layer. If you were to test code that uses the Solr search engine, you'd use another layer that starts and stops solr between tests. But most of the time you just use the default layers you copied from somewhere or that mr.bob gave you.

By convention, layers are defined in a module `testing` in your module root, ie `my.code.testing`. Your test classes should be in a folder named `tests`

Getting started

Mr.bob already created the testing layers. We will go through them now.

Next, it adds a method for testing that your add-on gets properly installed. This might seem stupid, but it isn't if you take into account that in plone land, things change with new releases. Having a GenericSetup profile installing Javascript files contains the assumption that the package wants a javascript file available in Plone. This assumption is explained in the syntax of the current Plone. By testing that the result is met, the Javascript file really is available, we spell out that assumption more clearly. The person that wants to make your package work 5 years from now, knows now that the result in his browser might be related to a missing file. Even if he does not understand the semantics from the old Plone on how to register js files, he has a good starting point on what to do to make this package compatible.

This is why it makes sense to write these tedious tests.

If nothing else matches, `test_setup.py` is the right location for anything GenericSetup related. In *Write Your Own Add-Ons to Customize Plone* we created a content type. It is time to test this.

We are going to create a test module named `test_talk`:

```python
from pkg_resources import resource_stream
from plone.app.testing import SITE_OWNER_NAME
from plone.app.testing import SITE_OWNER_PASSWORD
from plone.app.testing import TEST_USER_ID
from plone.app.testing import setRoles
from plone.dexterity.interfaces import IDexterityFTI
from plone.testing.z2 import Browser
from ploneconf.site.testing import PLONECONF_SITE_FUNCTIONAL_TESTING
from ploneconf.site.testing import PLONECONF_SITE_INTEGRATION_TESTING
from zope.component import createObject
from zope.component import queryUtility
import unittest


class TalkIntegrationTest(unittest.TestCase):

    layer = PLONECONF_SITE_INTEGRATION_TESTING

    def setUp(self):
        self.portal = self.layer['portal']
        setRoles(self.portal, TEST_USER_ID, ['Manager'])

    def test_fti(self):
        fti = queryUtility(IDexterityFTI, name='talk')
        self.assertTrue(fti)

    def test_schema(self):
        fti = queryUtility(IDexterityFTI, name='talk')
        schema = fti.lookupSchema()
        self.assertTrue(schema)
        # self.assertEqual(ITalk, schema)

    def test_factory(self):
        fti = queryUtility(IDexterityFTI, name='talk')
        factory = fti.factory
        talk = createObject(factory)
        # self.assertTrue(ITalk.providedBy(talk))
        self.assertTrue(talk)

    def test_adding(self):
        self.portal.invokeFactory('talk', 'talk')
        self.assertTrue(self.portal.talk)
        # self.assertTrue(ITalk.providedBy(self.portal.talk))


class TalkFunctionalTest(unittest.TestCase):

    layer = PLONECONF_SITE_FUNCTIONAL_TESTING

    def setUp(self):
        app = self.layer['app']
        self.portal = self.layer['portal']
        self.request = self.layer['request']
        self.portal_url = self.portal.absolute_url()

        # Set up browser
        self.browser = Browser(app)
        self.browser.handleErrors = False
        self.browser.addHeader(
            'Authorization',
            'Basic %s:%s' % (SITE_OWNER_NAME, SITE_OWNER_PASSWORD,)
        )

    def test_add_task(self):
        self.browser.open(self.portal_url + '/++add++talk')
        ctrl = self.browser.getControl
        ctrl(name="form.widgets.IDublinCore.title").value = "My Talk"
        ctrl(name="form.widgets.IDublinCore.description").value = \
            "This is my talk"
        ctrl(name="form.widgets.type_of_talk").value = ["Talk"]
        ctrl(name="form.widgets.details").value = "Long awesome talk"
        ctrl(name="form.widgets.audience:list").value = ["Advanced"]
        ctrl(name="form.widgets.speaker").value = "Team Banzai"
        ctrl(name="form.widgets.email").value = "banzai@example.com"
        img_ctrl = ctrl(name="form.widgets.image")
        img_ctrl.add_file(resource_stream(__name__, 'plone.png'),
                          'image/png', 'plone.png')
        ctrl(name="form.widgets.speaker_biography").value = \
            "Team Banzai is awesome, we are on Wikipedia!"
        ctrl("Save").click()

        talk = self.portal['my-talk']

        self.assertEqual('My Talk', talk.title)
        self.assertEqual('This is my talk',talk.description)
        self.assertEqual('Talk', talk.type_of_talk)
        self.assertEqual('Long awesome talk', talk.details.output)
        self.assertEqual({'Advanced'}, talk.audience)
        self.assertEqual('Team Banzai', talk.speaker)
        self.assertEqual((491, 128), talk.image.getImageSize())
        self.assertEqual('Team Banzai is awesome, we are on Wikipedia!',
                         talk.speaker_biography.output)

    def test_view_task(self):
        setRoles(self.portal, TEST_USER_ID, ['Manager'])
        self.portal.invokeFactory(
            "talk",
            id="my-talk",
            title="My Talk",
        )

        import transaction
        transaction.commit()

        self.browser.open(self.portal_url + '/my-talk')

        self.assertTrue('My Talk' in self.browser.contents)

```

In *Views I* we created a new view. We have to test this! This time, though, we are going to test it with a browser, too.

First, we add a simple test for the custom template in our Functional Test layer

```
    def test_custom_template(self):
        setRoles(self.portal, TEST_USER_ID, ['Manager'])
        self.portal.invokeFactory(
            "talk",
            id="my-talk",
            title="My Talk",
        )

        import transaction
        transaction.commit()

        self.browser.open(self.portal_url + '/training')

        self.assertIn('Dexterity for the win', self.browser.contents)
        self.assertIn('Deco is the future', self.browser.contents)
        self.assertIn('The State of Plone', self.browser.contents)
        self.assertIn('Diazo designs are great', self.browser.contents)
```

Exercise 1

We already wrote a talklistview and it is untested! We like to write unit tests first. But if you look at the Talklistview, you notice that you'd have to mock the portal_catalog, the context, and complex results from the catalog. I wrote earlier that it is ok to rewrite code to make it better testable. But in this example look at what you would test if you mocked everything mentioned above. You would test that your code iterates over a mocked list of mocked items, restructuring mocked attributes. There is not much sense in that. If you did some calculation, like ratings, things might look different, but not in this case.

We can write an integration test. We should test the good case, and edge cases. The simplest test we can write is a test where no talks exist.

Then we can create content. Looking through the code, we do not want the talks list to render results for documents. So add a a document. Also, the code does not want to render results for a document out of the current context. So create a folder and use this as a context. Then add a talk outside of this folder. The method iterates over audiences, make sure that you have at least one talk that has multiple audiences and check for that. Some advanced thing. Should you ever use an improved search system like collective.solr, results might get batched automatically. Check that if you have 101 talks, that you also get back 101 talks. Think about what you want to check in your results. Do you want to make a one to one comparison? How would you handle UUIDs?

A test creating 101 talks can be slow. It tests an edge case. There is a trick: create a new `TestCase` Class, and set an attribute `level` with the value of 2. This test will then only be run when you run the tests with the argument `-a 2` or `--all`

Solution

```
    def test_talklist(self):
        view = api.content.get_view(name='talklistview',
                                    context=self.portal,
                                    request=self.request)
        api.content.create(container=self.portal,
                           type='talk',
                           id='talk',
                           title='A Talk')
        talks = view.talks()
        self.assertEquals(1, len(talks))
        self.assertEquals(['start',
                           'audience',
                           'speaker',
                           'description',
                           'title',
                           'url',
                           'type_of_talk',
                           'room',
                           'uuid'],
                          talks[0].keys())

    def test_talklist_multipleaudiences(self):
        view = api.content.get_view(name='talklistview',
                                    context=self.portal,
                                    request=self.request)
        api.content.create(container=self.portal,
                           type='talk',
                           id='talk',
                           title='A Talk')
        self.portal.talk.audience = ['alpha', 'beta']
        notify(ObjectModifiedEvent(self.portal.talk))
        talks = view.talks()
        self.assertEquals(1, len(talks))
        self.assertEquals('alpha, beta', talks[0]['audience'])

    def test_talklist_filtering(self):
        api.content.create(container=self.portal,
                           type='talk',
                           id='talk',
                           title='A Talk')
        api.content.create(container=self.portal,
                           type='Folder',
                           id='talks-folder',
                           title='A talks Folder')
        api.content.create(container=self.portal['talks-folder'],
                           type='talk',
                           id='talk',
                           title='A Talk')
        api.content.create(container=self.portal['talks-folder'],
                           type='Document',
                           id='a Document',
                           title='A Document')
        view = api.content.get_view(name='talklistview',
                                    context=self.portal['talks-folder'],
                                    request=self.request)
        talks = view.talks()
        self.assertEquals(1, len(talks))
        self.assertEquals('A Talk', talks[0]['title'])


class SlowTalkIntegrationTest(unittest.TestCase):

    layer = PLONECONF_SITE_INTEGRATION_TESTING

    level = 2

    def setUp(self):
        self.portal = self.layer['portal']
        self.request = self.layer['request']
        setRoles(self.portal, TEST_USER_ID, ['Manager'])

    def test_talklist_many_results(self):
        view = api.content.get_view(name='talklistview',
                                    context=self.portal,
                                    request=self.request)
        for i in range(101):
            api.content.create(container=self.portal,
                               type='talk',
                               id='talk_{}'.format(i),
                               title='Talk {}'.format(i))
        talks = view.talks()
        self.assertEquals(101, len(talks))
        self.assertTrue(16, len(talks[-1]['uuid']))
```

Robot tests

Finally, we write a robot test:

```
# ============================================================================
# EXAMPLE ROBOT TESTS
# ============================================================================
#
# Run this robot test stand-alone:
#
#  $ bin/test -s plonetraining.testing -t test_talk.robot --all
#
# Run this robot test with robot server (which is faster):
#
# 1) Start robot server:
#
# $ bin/robot-server --reload-path src plonetraining.testing.testing.PLONETRAINING_
↪TESTING_ACCEPTANCE_TESTING
#
# 2) Run robot tests:
#
# $ bin/robot src/plonetraining/testing/tests/robot/test_talk.robot
#
# See the http://docs.plone.org for further details (search for robot
# framework).
#
# ============================================================================

*** Settings *****************************************************************

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

Library  Remote  ${PLONE_URL}/RobotRemote

Test Setup  Open test browser
Test Teardown  Close all browsers


*** Test Cases ***************************************************************

Scenario: As a site administrator I can add a Talk
  Given a logged-in site administrator
    and an add talk form
   When I type 'My Talk' into the title field
    and I type 'Awesome talk' into the details field
    and I type 'Team Banzai' into the speakers field
    and I type 'banzai@example.com' into the email field
    and I submit the form
   Then a talk with the title 'My Talk' has been created

```

```
Scenario: As a site administrator I can view a Talk
  Given a logged-in site administrator
    and a talk 'My Talk'
   When I go to the talk view
   Then I can see the talk title 'My Talk'

Scenario: As a visitor I can view the new talk list
   When I go to the talk list view
   Then I can see a talk about 'Diazo designs are great'


*** Keywords *****************************************************************

# --- Given ------------------------------------------------------------------

a logged-in site administrator
  Enable autologin as  Site Administrator

an add talk form
  Go To  ${PLONE_URL}/++add++talk

a talk 'My Talk'
  Create content  type=talk  id=my-talk  title=My Talk


# --- WHEN -------------------------------------------------------------------

I type '${title}' into the title field
  Input Text  name=form.widgets.IDublinCore.title  ${title}

I type '${details}' into the details field
  Select frame  form-widgets-details_ifr
  Input text  tinymce  ${details}
  Unselect Frame

I type '${speaker}' into the speakers field
  Input Text  name=form.widgets.speaker  ${speaker}

I type '${email}' into the email field
  Input Text  name=form.widgets.email  ${email}

I submit the form
  Click Button  Save

I go to the talk view
  Go To  ${PLONE_URL}/my-talk
  Wait until page contains  Site Map

I go to the talk list view
  Go To  ${PLONE_URL}/demoview
  Wait until page contains  Site Map


# --- THEN -------------------------------------------------------------------

a talk with the title '${title}' has been created
  Wait until page contains  Site Map
  Page should contain  ${title}
  Page should contain  Item created

I can see the talk title '${title}'
  Wait until page contains  Site Map
  Page should contain  ${title}

I can see a talk about '${topic}'
  Wait until page contains  Site Map
  Page should contain  ${topic}
```

When you run your tests, you might notice that the robot tests didn't run. This is a feature activated by the robot layer, because robot tests can be quite slow. If you run your tests with **`./bin/test --all`** your robot tests will run. Now you will realize that you cannot work any more because a browser window pops up all the time.

There are 3 possible workarounds:

- install the headless browser, Phantomjs. Then run the tests with an environment variable **`ROBOT_BROWSER=phantomjs bin/test --all`** This did not work for me btw.
- Install **`xvfb`**, a framebuffer. You wont see the browser then. After installing, start xvfb like this: **`Xvfb :99.0 -screen 0 1024x768x24`**. Then run your tests, declaring to connect to the non-default X Server: **`DISPLAY=:99.0 bin/test --all`**
- Install Xephyr, it is also a framebuffer, but visible in a window. Start it the same way as you start Xvfb.

The first method, with Phantomjs, will throw failures with our tests, unfortunately.

For debugging, you can run the test like this **`ROBOT_SELENIUM_RUN_ON_FAILURE=Debug bin/test --all`**. This will stop the test at the first failure and you end up in an interactive shell where you can try various Robot Framework commands.

More information

For more in-depth information and reference see

- plone.app.testing documentation.
- plone.testing package

Behaviors

Get the code!

Get the code for this chapter (More info):

```
git checkout behaviors_1
```

In this part you will:

- Add another field to talks by using a behavior

Topics covered:

- Behaviors

You can extend the functionality of your dexterity object by writing an adapter that adapts your dexterity object to add another feature or aspect.

But if you want to use this adapter, you must somehow know that an object implements that. Also, adding more fields to an object would not be easy with such an approach.

Dexterity Approach

Dexterity has a solution for it, with special adapters that are called and registered by the name behavior.

A behavior can be added to any content type through the web and at runtime.

All default views (e.g. the add- and edit-forms) know about the concept of behaviors and when rendering forms, the views also check whether there are behaviors referenced with the current context and if these behaviors have a schema of their own, these fields get shown in addition.

Names and Theory

The name behavior is not a standard term in software development. But it is a good idea to think of a behavior as an aspect. You are adding an aspect to your content type and you want to write your aspect in such a way that it works independently of the content type on which the aspect is applied. You should not have dependencies to specific fields of your object or to other behaviors.

Such an object allows you to apply the Open/closed principle to your dexterity objects.

Practical example

So, let us write our own small behavior.

In the future, we want our presentation to be represented in Lanyrd (a Social Conference Directory - Lanyrd.com) too. For now we will just provide a link so that visitors can collaborate easily with the Lanyrd site.

So for now, our behavior just adds a new field for storing the url to Lanyrd.

We want to keep a clean structure, so we create a `behaviors` directory first, and include it into the zcml declarations of our `configure.zcml`.

```
<include package=".behaviors" />
```

Then, we add an empty `behaviors/__init__.py` and a `behaviors/configure.zcml` containing

Advanced reference

It can be a bit confusing when to use factories or marker interfaces and when not to.

If you do not define a factory, your attributes will be stored directly on the object. This can result in clashes with other behaviors.

You can avoid this by using the `plone.behavior.AnnotationStorage` factory. This stores your attributes in an Annotation. But then you *must* use a marker interface if you want to have custom viewlets, browser views or portlets.

Without it, you would have no interface against which you could register your views.

```
<configure
    xmlns="http://namespaces.zope.org/zope"
    xmlns:plone="http://namespaces.plone.org/plone"
    i18n_domain="ploneconf.site">
```

```

  <plone:behavior
      title="Social Behavior"
      description="Adds a link to lanyrd"
      provides=".social.ISocial"
      />

</configure>
```

And a behaviors/social.py containing:

```
# -*- coding: utf-8 -*-
from plone.autoform.interfaces import IFormFieldProvider
from plone.supermodel import directives
from plone.supermodel import model
from zope import schema
from zope.interface import alsoProvides


class ISocial(model.Schema):

    directives.fieldset(
        'social',
        label=u'Social',
        fields=('lanyrd',),
    )

    lanyrd = schema.URI(
        title=u"Lanyrd link",
        description=u"Add URL",
        required=False,
    )

alsoProvides(ISocial, IFormFieldProvider)
```

Let’s go through this step by step.

1. We register a behavior in *behaviors/configure.zcml*. We do not say for which content type this behavior is valid. You do this through the web or in the GenericSetup profile.
2. We create a marker interface in *behaviors/social.py* for our behavior and make it also a schema containing the fields we want to declare. We could just define schema fields on a zope.interface class, but we use an extended form from plone.supermodel, else we could not use the fieldset features.
3. We also add a fieldset so that our fields are not mixed with the normal fields of the object.
4. We add a normal URI schema field to store the URI to lanyrd.
5. We mark our schema as a class that also implements the IFormFieldProvider interface. This is a marker interface, we do not need to implement anything to provide the interface.

Adding it to our talk

We could add this behavior now via the plone control panel. But instead, we will do it directly and properly in our GenericSetup profile

We must add the behavior to profiles/default/types/talk.xml:

```xml
<?xml version="1.0"?>
<object name="talk" meta_type="Dexterity FTI" i18n:domain="plone"
   xmlns:i18n="http://xml.zope.org/namespaces/i18n">
   ...
 <property name="behaviors">
  <element value="plone.app.dexterity.behaviors.metadata.IDublinCore"/>
  <element value="plone.app.content.interfaces.INameFromTitle"/>
  <element value="ploneconf.site.behaviors.social.ISocial"/>
 </property>
 ...
</object>
```

Writing Viewlets

Get the code!

Get the code for this chapter (More info):

```
git checkout viewlets_1
```

In this part you will:

- Display data from a behavior in a viewlet

Topics covered:

- Viewlets

A viewlet for the social behavior

A viewlet is not a view but a snippet of HTML and logic that can be put in various places in the site. These places are called `viewletmanager`.

- Inspect existing viewlets and their managers by going to http://localhost:8080/Plone/@@manage-viewlets.
- We already customized a viewlet (`colophon.pt`). Now we add a new one.
- Viewlets don't save data (portlets do)
- Viewlets have no user interface (portlets do)

Social viewlet

Let's add a link to the site that uses the information that we collected using the social behavior.

We register the viewlet in `browser/configure.zcml`.

```xml
<browser:viewlet
  name="social"
  for="ploneconf.site.behaviors.social.ISocial"
  manager="plone.app.layout.viewlets.interfaces.IBelowContentTitle"
  class=".viewlets.SocialViewlet"
  layer="zope.interface.Interface"
  template="templates/social_viewlet.pt"
  permission="zope2.View"
  />
```

for, manager, layer and permission are constraints that limit the contexts in which the viewlet is loaded and rendered, by filtering out all the contexts that do not match those constraints.

This registers a viewlet called social. It is visible on all content that implements the interface ISocial from our behavior. It is also good practice to bind it to a specific layer, so it only shows up if our add-on is actually installed. We will return to this in a later chapter.

The viewlet class SocialViewlet is expected in a file browser/viewlets.py.

```
from plone.app.layout.viewlets import ViewletBase

class SocialViewlet(ViewletBase):
    pass
```

This class does nothing except rendering the associated template (That we have yet to write)

Let's add the missing template templates/social_viewlet.pt.

```
<div id="social-links">
    <a href="#"
       class="lanyrd-link"
       tal:define="link view/lanyrd_link"
       tal:condition="link"
       tal:attributes="href link">
         See this talk on Lanyrd!
    </a>
</div>
```

As you can see this is not a valid HTML document. That is not needed, because we don't want a complete view here, just a html snippet.

There is a tal:define statement, querying for view/lanyrd_link. Same as for views, viewlets have access to their class in page templates, as well.

We have to extend the Social Viewlet now to add the missing attribute:

Why not to access context directly

In this example, ISocial(self.context) does return the context directly. It is still good to use this idiom for two reasons:

1. It makes it clear that we only want to use the ISocial aspect of the object
2. If we decide to use a factory, for example to store our attributes in an annotation, we would *not* get back our context, but the adapter.

Therefore in this example you could simply write return self.context.lanyrd.

```
from plone.app.layout.viewlets import ViewletBase
from ploneconf.site.behaviors.social import ISocial

class SocialViewlet(ViewletBase):

    def lanyrd_link(self):
        adapted = ISocial(self.context)
        return adapted.lanyrd
```

So far, we

- register the viewlet to content that has the ISocial Interface.
- adapt the object to its behavior to be able to access the fields of the behavior
- return the link

Exercise 1

Register a viewlet 'number_of_talks' in the footer that is only visible to admins (the permission you are looking for is `cmf.ManagePortal`). Use only a template (no class) to display the number of talks already submitted. Hint: Use Acquisition to get the catalog (You know, you should not do this but there is plenty of code out there that does it...)

Solution

Register the viewlet in `browser/configure.zcml`

```
<browser:viewlet
  name="number_of_talks"
  for="*"
  manager="plone.app.layout.viewlets.interfaces.IPortalFooter"
  layer="zope.interface.Interface"
  template="templates/number_of_talks.pt"
  permission="cmf.ManagePortal"
  />
```

For the `for` and `layer`-parameters `*` is shorthand for `zope.interface.Interface` and the same effect as omitting them: The viewlet will be shown for all types of pages and for all Plone sites within your Zope instance.

Add the template `browser/templates/number_of_talks.pt`:

```
<div class="number_of_talks"
     tal:define="catalog python:context.portal_catalog;
                 talks python:len(catalog(portal_type='talk'));">
    There are <span tal:replace="talks" /> talks.
</div>
```

`python:context.portal_catalog` will return the catalog through Acquisition. Be careful if you want to use path expressions: `content/portal_catalog` calls the catalog (and returns all brains). You need to prevent this by using `nocall:content/portal_catalog`.

Relying on Acquisition is a bad idea. It would be much better to use the helper view `plone_tools` from `plone/app/layout/globals/tools.py` to get the catalog.

```
<div class="number_of_talks"
     tal:define="catalog context/@@plone_tools/catalog;
                 talks python:len(catalog(portal_type='talk'));">
    There are <span tal:replace="talks" /> talks.
</div>
```

`context/@@plone_tools/catalog` traverses to the view `plone_tools` and calls its method `catalog()`. In python it would look like this:

```
<div class="number_of_talks"
     tal:define="catalog python:context.restrictedTraverse('plone_tools').catalog();
                 talks python:len(catalog(portal_type='talk'));">
    There are <span tal:replace="talks" /> talks.
</div>
```

It is not a good practice to query the catalog within a template since even simple logic like this should live in Python. But it is very powerful if you are debugging or need a quick and dirty solution.

In Plone 5 you could even write it like this:

```
<?python

from plone import api
catalog = api.portal.get_tool('portal_catalog')
talks_amount = len(catalog(portal_type='talk'))

?>

<div class="number_of_talks">
    There are ${talks_amount} talks.
</div>
```

Exercise 2

Register a viewlet 'days_to_conference' in the header. Use a class and a template to display the number of days until the conference. You get bonus points if you display it in a nice format (think "In 2 days" and "Last Month") by using either javascript or a python library.

Solution

In `configure.zcml`:

```
<browser:viewlet
  name="days_to_conference"
  for="*"
  manager="plone.app.layout.viewlets.interfaces.IPortalHeader"
  layer="*"
  class=".viewlets.DaysToConferenceViewlet"
  template="templates/days_to_conference.pt"
  permission="zope2.View"
  />
```

In `viewlets.py`:

```
from plone.app.layout.viewlets import ViewletBase
from datetime import datetime
import arrow

CONFERENCE_START_DATE = datetime(2015, 10, 12)


class DaysToConferenceViewlet(ViewletBase):

    def date(self):
        return CONFERENCE_START_DATE

    def human(self):
        return arrow.get(CONFERENCE_START_DATE).humanize()
```

Setting the date in python is not very user-friendly. In the chapter *Manage Settings with Registry, Controlpanels and Vocabularies* you learn how store global configuration and easily create controlpanels.

And in `templates/days_to_conference.pt`:

```
<div class="days_to_conf">
    ${python: view.human()}
</div>
```

Or using the moment pattern in Plone 5:

```
<div class="pat-moment"
     data-pat-moment="format: relative">
    ${python: view.date()}
</div>
```

Programming Plone

In this part you will:

- Learn about the right ways to do something in code in Plone.
- Learn to debug

Topics covered:

- Debugging
- Plone API
- Portal tools

plone.api

The most important tool nowadays for plone developers is the add-on plone.api that covers 20% of the tasks any Plone developer does 80% of the time. If you are not sure how to handle a certain task be sure to first check if plone.api has a solution for you.

The API is divided in five sections. Here is one example from each:

- *Content:* Create content
- *Portal:* Send E-Mail
- *Groups:* Grant roles to group
- *Users:* Get user roles
- *Environment:* Switch roles inside a block

plone.api is a tool for integrators and developers that is included when you install Plone, though for technical reasons it is not used by Plone itself.

In existing code you'll often encounter methods that don't mean anything to you. You'll have to use the source to find out what they do.

Some of these methods will be replaced by plone.api in the future:

- `Products.CMFCore.utils.getToolByName()` -> `api.portal.get_tool()`
- `zope.component.getMultiAdapter()` -> `api.content.get_view()`

portal-tools

Some parts of Plone are very complex modules in themselves (e.g. the versioning machinery of `Products.CMFEditions`). Some of them have an API that you will have to learn sooner or later.

Here are a few examples:

portal_catalog `unrestrictedSearchResults()` returns search results without checking if the current user has the permission to access the objects.

`uniqueValuesFor()` returns all entries in an index

portal_setup `runAllExportSteps()` generates a tarball containing artifacts from all export steps.

portal_quickinstaller `isProductInstalled()` checks if a product is installed.

Usually the best way to learn about the API of a tool is to look in the `interfaces.py` in the respective package and read the docstrings.

Debugging

Here are some tools and techniques we often use when developing and debugging. We use some of them in various situations during the training.

tracebacks and the log The log (and the console when running in foreground) collects all log messages Plone prints. When an exception occurs Plone throws a traceback. Most of the time the traceback is everything you need to find out what is going wrong. Also adding your own information to the log is very simple.

pdb The python debugger pdb is the single most important tool for us when programming. Just add `import pdb; pdb.set_trace()` in your code and debug away!

Since Plone 5 you can even add it to templates: add `<?python import pdb; pdb.set_trace() ?>` to a template and you end up in a pdb shell on calling the template. Look at the variable `econtext` to see what might have gone wrong.

ipdb Enhanced pdb with the power of IPython, e.g. tab completion, syntax highlighting, better tracebacks and introspection. It also works nicely with `Products.PDBDebugMode`.

Products.PDBDebugMode An add-on that has two killer features.

Post-mortem debugging: throws you in a pdb whenever an exception occurs. This way you can find out what is going wrong.

pdb view: simply adding `/pdb` to a url drops you in a pdb session with the current context as `self.context`. From there you can do just about anything.

Debug mode When starting Plone using **`./bin/instance debug`** you'll end up in an interactive debugger.

plone.app.debugtoolbar An add-on that allows you to inspect nearly everything. It even has an interactive console, a tester for TALES-expressions and includs a reload-feature like `plone.reload`.

plone.reload An add-on that allows to reload code that you changed without restarting the site. It is also used by `plone.app.debugtoolbar`.

Products.PrintingMailHost An add-on that prevents Plone from sending mails. Instead, they are logged.

Products.enablesettrace or Products.Ienablesettrace Add-on that allows to use pdb and ipdb in Python skin scripts. Very useful when debugging legacy code.

`verbose-security = on` An option for the recipe `plone.recipe.zope2instance` that logs the detailed reasons why a user might not be authorized to see something.

`./bin/buildout annotate` An option when running buildout that logs all the pulled packages and versions.

Sentry Sentry is an error logging application you can host yourself. It aggregates tracebacks from many sources and (here comes the killer feature) even the values of variables in the traceback. We use it in all our production sites.

zopepy Buildout can create a python shell for you that has all the packages from your Plone site in its python path. Add the part like this:

```
[zopepy]
recipe = zc.recipe.egg
eggs = ${instance:eggs}
interpreter = zopepy
```

IDEs and Editors

In this part you will:

- Learn about Editors

Topics covered:

- Many editors

Plone consists of more than 20.000 files! You need a tool to manage that. No development environment is complete without a good editor.

People pick editors themselves. Use whatever you are comfortable and productive with. These are the most used editors in the Plone community:

- Sublime
- PyCharm
- Wing IDE
- PyDev for Eclipse
- Aptana Studio
- vim
- emacs
- Textmate

Some features that most editors have in one form or another are essential when developing with Plone.

- **Find in project** (SublimeText 3: `cmd + shift + f`)
- **Find files in Project** (SublimeText 3: `cmd + t`)
- **Find methods and classes in Project** (SublimeText 3: `cmd + shift + r`)
- **Goto Definition** (SublimeText3 with codeintel: `alt + click`)
- **Powerful search & replace**

The capability of performing a *full text search* through the complete Plone code is invaluable. Thanks to omelette, an SSD and plenty of RAM you can search through the complete Plone code base in 3 seconds.

Note: Some Editors/IDE's have to be extend to be fully featured. Here are some packages we recommend when using Sublime Text 3:

- SublimeCodeIntel (Goto Definition)
- BracketHighlighter
- GitGutter
- FileDiffs
- SublimeLinter with SublimeLinter-flake8 ...
- INI (syntax for ini-Files)
- SideBarEnhancements
- MacTerminal
- SyncedSideBar

Dexterity Types II: Growing Up

Get the code!

Get the code for this chapter (More info):

```
git checkout dexterity_2
```

The existing talks are still lacking some functionality we want to use.

In this part we will:

- add a marker interface to our talk type,
- create custom catalog indexes,
- query the catalog for them,
- enable some more default features for our type.

Add a marker interface to the talk type

Marker Interfaces

The content type *Talk* is not yet a *first class citizen* because it does not implement its own interface. Interfaces are like nametags, telling other elements who and what you are and what you can do. A marker interface is like such a nametag. The talks actually have an auto-generated marker interface `plone.dexterity.schema.generated.Plone_0_talk`.

One problem is that the name of the Plone instance `Plone` is part of that interface name. If you now moved these types to a site with another name the code that uses these interfaces would no longer find the objects in question.

To create a real name-tag we add a new `Interface` to `interfaces.py`:

```python
# -*- coding: utf-8 -*-
"""Module where all interfaces, events and exceptions live."""

from zope.publisher.interfaces.browser import IDefaultBrowserLayer
from zope.interface import Interface


class IPloneconfSiteLayer(IDefaultBrowserLayer):
    """Marker interface that defines a browser layer."""


class ITalk(Interface):
    """Marker interface for Talks"""
```

`ITalk` is a marker interface. We can bind Views and Viewlets to content that provide these interfaces. Lets see how we can provide this Interface. There are two solutions for this.

1. Let them be instances of a class that implements this Interface.
2. Register this interface as a behavior and enable it on talks.

The first option has an important drawback: only *new* talks would be instances of the new class. We would either have to migrate the existing talks or delete them.

So let's register the interface as a behavior in `behaviors/configure.zcml`

```xml
<plone:behavior
    title="Talk"
    description="Marker interface for talks to be able to bind views to."
    provides="..interfaces.ITalk"
    />
```

And enable it on the type in `profiles/default/types/talk.xml`

```xml
<property name="behaviors">
 <element value="plone.app.dexterity.behaviors.metadata.IDublinCore"/>
 <element value="plone.app.content.interfaces.INameFromTitle"/>
 <element value="ploneconf.site.behaviors.social.ISocial"/>
 <element value="ploneconf.site.interfaces.ITalk"/>
</property>
```

Either reinstall the add-on, apply the behavior by hand or run an upgrade step (see below) and the interface will be there.

Then we can safely bind the `talkview` to the new marker interface.

```xml
<browser:page
    name="talkview"
    for="ploneconf.site.interfaces.ITalk"
    layer="zope.interface.Interface"
    class=".views.TalkView"
    template="templates/talkview.pt"
    permission="zope2.View"
    />
```

Now the `/talkview` can only be used on objects that implement said interface. We can now also query the catalog for objects providing this interface `catalog(object_provides="ploneconf.site.interfaces.ITalk")()`. The `talklistview` and the `demoview` do not get this constraint since they are not only used on talks.

Note: Just for completeness sake, this is what would have to happen for the first option (associating the ITalk interface with a Talk class):

- Create a new class that inherits from plone.dexterity.content.Container and implements the marker interface.

```
from plone.dexterity.content import Container
from ploneconf.site.interfaces import ITalk
from zope.interface import implementer

@implementer(ITalk)
class Talk(Container):
    """Class for Talks"""
```

- Modify the class for new talks in profiles/default/types/talk.xml

```
...
<property name="add_permission">cmf.AddPortalContent</property>
<property name="klass">ploneconf.site.content.talk.Talk</property>
<property name="behaviors">
...
```

- Create an upgrade step that changes the class of the existing talks. A reuseable method to do such a thing is in plone.app.contenttypes.migration.dxmigration.migrate_base_class_to_new_class.

Upgrade steps

When projects evolve you sometimes want to modify various things while the site is already up and brimming with content and users. Upgrade steps are pieces of code that run when upgrading from one version of an add-on to a newer one. They can do just about anything. We will use an upgrade-step to enable the new behavior instead of reinstalling the addon.

We will create an upgrade step that:

- runs the typeinfo step (i.e. loads the GenericSetup configuration stored in profiles/default/types.xml and profiles/default/types/... so we don't have to reinstall the add-on to have our changes from above take effect) and
- cleans up the talks that might be scattered around the site in the early stages of creating it. We will move all talks to a folder talks (unless they already are there).

Upgrade steps can be registered in their own ZCML file to prevent cluttering the main configure.zcml. Include a new upgrades.zcml in our configure.zcml by adding:

```
<include file="upgrades.zcml" />
```

Create upgrades.zcml:

```
<configure
  xmlns="http://namespaces.zope.org/zope"
  xmlns:i18n="http://namespaces.zope.org/i18n"
  xmlns:genericsetup="http://namespaces.zope.org/genericsetup"
  i18n_domain="ploneconf.site">

  <genericsetup:upgradeStep
      title="Update and cleanup talks"
      description="Update typeinfo and move talks to a folder 'talks'"
      source="1000"
      destination="1001"
      handler="ploneconf.site.upgrades.upgrade_site"
      sortkey="1"
      profile="ploneconf.site:default"
      />

</configure>
```

The upgrade step bumps the version number of the GenericSetup profile of `ploneconf.site` from 1000 to 1001. The version is stored in `profiles/default/metadata.xml`. Change it to

```
<version>1001</version>
```

GenericSetup now expects the code as a method `upgrade_site()` in the file `upgrades.py`. Let's create it.

```
# -*- coding: utf-8 -*-
from plone import api

import logging

default_profile = 'profile-ploneconf.site:default'
logger = logging.getLogger(__name__)


def upgrade_site(setup):
    setup.runImportStepFromProfile(default_profile, 'typeinfo')
    portal = api.portal.get()
    # Create a folder 'The event' if needed
    if 'the-event' not in portal:
        event_folder = api.content.create(
            container=portal,
            type='Folder',
            id='the-event',
            title=u'The event')
    else:
        event_folder = portal['the-event']

    # Create folder 'Talks' inside 'The event' if needed
    if 'talks' not in event_folder:
        talks_folder = api.content.create(
            container=event_folder,
            type='Folder',
            id='talks',
            title=u'Talks')
    else:
        talks_folder = event_folder['talks']
    talks_url = talks_folder.absolute_url()

    # Find all talks
    brains = api.content.find(portal_type='talk')
    for brain in brains:
        if talks_url in brain.getURL():
            # Skip if the talk is already somewhere inside the target-folder
            continue
        obj = brain.getObject()
        logger.info('Moving {} to {}'.format(
            obj.absolute_url(), talks_folder.absolute_url()))
        # Move talk to the folder '/the-event/talks'
        api.content.move(
            source=obj,
            target=talks_folder,
            safe_id=True)
```

Note:

- Upgrade-steps get the tool `portal_setup` passed as their argument.
- The `portal_setup` tool has a method `runImportStepFromProfile()`
- We create the needed folder-structure if it does not exists.

After restarting the site we can run the step:

- Go to the *Add-ons* control panel http://localhost:8080/Plone/prefs_install_products_form. There should now be a new section **Upgrades** and a button to upgrade from 1000 to 1001.
- Run the upgrade step by clicking on it.

On the console you should see logging messages like:

```
INFO ploneconf.site.upgrades Moving http://localhost:8080/Plone/old-talk1 to http://localhost:8080/Plone/the-event/talks
```

Alternatively you also select which upgrade steps to run like this:

- In the ZMI go to *portal_setup*
- Go to the tab *Upgrades*
- Select *ploneconf.site* from the dropdown and click *Choose profile*
- Run the upgrade step.

See also:

http://docs.plone.org/develop/addons/components/genericsetup.html#id1

Note: Upgrading from an older version of Plone to a newer one also runs upgrade steps from the package `plone.app.upgrade`. You should be able to upgrade a clean site from 2.5 to 5.0 with one click.

For an example see the upgrade-step to Plone 5.0a1 https://github.com/plone/plone.app.upgrade/blob/master/plone/app/upgrade/v50/alphas.py#L37

Add a browserlayer

A browserlayer is another such marker interface. Browserlayers allow us to easily enable and disable views and other site functionality based on installed add-ons and themes.

Since we want the features we write only to be available when `ploneconf.site` actually is installed we can bind them to a browserlayer.

Our package already has a browserlayer (added by `bobtemplates.plone`). See `interfaces.py`:

```
# -*- coding: utf-8 -*-
"""Module where all interfaces, events and exceptions live."""

```

```
from zope.publisher.interfaces.browser import IDefaultBrowserLayer
from zope.interface import Interface


class IPloneconfSiteLayer(IDefaultBrowserLayer):
    """Marker interface that defines a browser layer."""


class ITalk(Interface):
    """Marker interface for Talks"""
```

It is enabled by GenericSetup when installing the package since it is registered in the `profiles/default/browserlayer.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<layers>
  <layer
      name="ploneconf.site"
      interface="ploneconf.site.interfaces.IPloneconfSiteLayer"
      />
</layers>
```

We should bind all views to it. Here is an example using the talkview.

```
<browser:page
    name="talklistview"
    for="*"
    layer="..interfaces.IPloneconfSiteLayer"
    class=".views.TalkListView"
    template="templates/talklistview.pt"
    permission="zope2.View"
    />
```

Note the relative Python path `interfaces.IPloneconfSiteLayer`. It is equivalent to the absolute path `ploneconf.site.interfaces.IPloneconfSiteLayer`.

See also:

http://docs.plone.org/develop/plone/views/layers.html

Exercise

Do you need to bind the *Social viewlet* from the chapter 'Writing Viewlets' to this new browser layer?

Solution

No, it would make no difference since the viewlet is already bound to the marker interface `ploneconf.site.behaviors.social.ISocial`.

Add catalog indexes

In the `talklistview` we had to wake up all objects to access some of their attributes. That is OK if we don't have many objects and they are light dexterity objects. If we had thousands of objects this might not be a good idea.

Instead of loading them all into memory we will use catalog indexes to get the data we want to display.

Add a new file `profiles/default/catalog.xml`

```
<?xml version="1.0"?>
<object name="portal_catalog">
  <index name="type_of_talk" meta_type="FieldIndex">
    <indexed_attr value="type_of_talk"/>
  </index>
  <index name="speaker" meta_type="FieldIndex">
    <indexed_attr value="speaker"/>
  </index>
  <index name="audience" meta_type="KeywordIndex">
    <indexed_attr value="audience"/>
  </index>

  <column value="audience" />
  <column value="type_of_talk" />
  <column value="speaker" />
</object>
```

This adds new indexes for the three fields we want to show in the listing. Note that *audience* is a `KeywordIndex` because the field is multi-valued, but we want a separate index entry for every value in an object.

The `column ..` entries allow us to display the values of these indexes in the tableview of collections.

Note: Until Plone 4.3.2 adding indexes in `catalog.xml` was harmful because reinstalling the add-on purged the indexes! See https://www.starzel.de/blog/a-reminder-about-catalog-indexes.

- Reinstall the add-on
- Go to http://localhost:8080/Plone/portal_catalog/manage_catalogAdvanced to update the catalog
- Go to http://localhost:8080/Plone/portal_catalog/manage_catalogIndexes to inspect and manage the new indexes

See also:

http://docs.plone.org/develop/plone/searching_and_indexing/indexing.html

Note: The new indexes are still empty. We'll have to reindex them. To do so by hand go to http://localhost:8080/Plone/portal_catalog/manage_catalogIndexes, select the new indexes and click *Reindex*. We could also rebuild the whole catalog by going to the *advanced*-tab and clicking *Clear and Rebuild*. For large sites that can take a long time.

We could also write an upgrade step to enable the catalog-indexes and reindex all talks:

```
def add_some_indexes(setup):
    setup.runImportStepFromProfile(default_profile, 'catalog')
    for brain in api.content.find(portal_type='talk'):
        obj = brain.getObject()
        obj.reindexObject(idxs=['type_of_talk', 'speaker', 'audience'])
```

Query for custom indexes

The new indexes behave like the ones that Plone has already built in:

```
>>> (Pdb) from Products.CMFCore.utils import getToolByName
>>> (Pdb) catalog = getToolByName(self.context, 'portal_catalog')
>>> (Pdb) catalog(type_of_talk='Keynote')
[<Products.ZCatalog.Catalog.mybrains object at 0x10737b9a8>, <Products.ZCatalog.
↪Catalog.mybrains object at 0x10737b9a8>]
>>> (Pdb) catalog(audience=('Advanced', 'Professionals'))
[<Products.ZCatalog.Catalog.mybrains object at 0x10737b870>, <Products.ZCatalog.
↪Catalog.mybrains object at 0x10737b940>, <Products.ZCatalog.Catalog.mybrains object␣
↪at 0x10737b9a8>]
>>> (Pdb) brain = catalog(type_of_talk='Keynote')[0]
>>> (Pdb) brain.speaker
u'David Glick'
```

We now can use the new indexes to improve the talklistview so we don't have to *wake up* the objects any more. Instead we use the brains' new attributes.

```
class TalkListView(BrowserView):
    """ A list of talks
    """

    def talks(self):
        results = []
        brains = api.content.find(context=self.context, portal_type='talk')
        for brain in brains:
            results.append({
                'title': brain.Title,
                'description': brain.Description,
                'url': brain.getURL(),
                'audience': ', '.join(brain.audience or []),
                'type_of_talk': brain.type_of_talk,
                'speaker': brain.speaker,
                'uuid': brain.UID,
                })
        return results
```

The template does not need to be changed and the result in the browser did not change, either. But when listing a large number of objects the site will now be faster since all the data you use comes from the catalog and the objects do not have to be loaded into memory.

Add collection criteria

To be able to search content in collections using these new indexes we would have to register them as criteria for the querystring widget that collections use. As with all features make sure you only do this if you really need it!

Add a new file `profiles/default/registry.xml`

```
<registry>
  <records interface="plone.app.querystring.interfaces.IQueryField"
           prefix="plone.app.querystring.field.audience">
    <value key="title">Audience</value>
    <value key="description">A custom speaker 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>
  <records interface="plone.app.querystring.interfaces.IQueryField"
           prefix="plone.app.querystring.field.type_of_talk">
    <value key="title">Type of Talk</value>
    <value key="description">A custom 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>
  <records interface="plone.app.querystring.interfaces.IQueryField"
           prefix="plone.app.querystring.field.speaker">
    <value key="title">Speaker</value>
    <value key="description">A custom 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>
```

See also:

http://docs.plone.org/develop/plone/functionality/collections.html#add-new-collection-criteria-new-style-plone-app-collection-installe

Add versioning through GenericSetup

Configure the versioning policy and a diff-view for talks through GenericSetup.

Add new file `profiles/default/repositorytool.xml`

```
<?xml version="1.0"?>
<repositorytool>
  <policymap>
    <type name="talk">
      <policy name="at_edit_autoversion"/>
      <policy name="version_on_revert"/>
    </type>
  </policymap>
</repositorytool>
```

Add new file `profiles/default/diff_tool.xml`

```
<?xml version="1.0"?>
<object>
  <difftypes>
    <type portal_type="talk">
      <field name="any" difftype="Compound Diff for Dexterity types"/>
    </type>
  </difftypes>
</object>
```

Finally you need to activate the versioning behavior on the content type. Edit `profiles/default/types/talk.xml`:

```
<property name="behaviors">
 <element value="plone.app.dexterity.behaviors.metadata.IDublinCore"/>
 <element value="plone.app.content.interfaces.INameFromTitle"/>
 <element value="ploneconf.site.behaviors.social.ISocial"/>
 <element value="ploneconf.site.interfaces.ITalk"/>
 <element value="plone.app.versioningbehavior.behaviors.IVersionable" />
</property>
```

Note: There is currently a bug that breaks showing diffs when multiple-choice fields were changed.

Summary

The talks are now grown up:

- They provide a interface to which you can bind features like views
- Some fields are indexed in the catalog making the listing faster
- Talks are now versioned
- You wrote your first upgrade-step to move the talks around: Whopee!

Custom Search

If the chapters about views seem complex, the custom search add-ons shown below might be a great alternative until you feel comfortable writing views and templates. Here are two addons that allow you to add custom searches and content listings through the web in Plone.

eea.facetednavigation

eea.facetednavigation is a full-featured and a very powerful addon to improve search within large collections of items. No programming skills are required to configure it since the configuration is done TTW. It lets you gradually select and explore different facets (metadata/properties) of the site content and narrow down you search quickly and dynamically.

- Install eea.facetednavigation
- Enable it on a new folder “Discover talks” by clicking on *Actions* > *Enable faceted navigation*.
- Click on the *Faceted* > *Configure* to configure it through the web.
  - Select ‘Talk’ for *Portal type*, hide *Results per page*
  - Add a checkboxes widget to the left and use the catalog index *Audience* for it.
  - Add a select widget for speaker
  - Add a radio widget for type_of_talk

Examples:

- http://www.dipf.de/en/research/projects
- https://www.mountaineers.org/learn/courses-clinics-seminars
- https://www.dyna-jet.com/hochdruckreiniger

See also:

We use the new catalog indexes to provide the data for the widgets and search the results. For other use cases we could also use either the built-in vocabularies (https://pypi.python.org/pypi/plone.app.vocabularies) or create custom vocabularies for this.

- Custom vocabularies ttw using Products.ATVocabularyManager
- Programming using Vocabularies: http://docs.plone.org/external/plone.app.dexterity/docs/advanced/vocabularies.html

collective.portlet.collectionfilter

A more light-weight solution for custom searches and faceted navigation is collective.portlet.collectionfilter. By default it allows you to search among the results of a collection and/or filter the results by keywords, author or type. It can also be extended quite easily to allow additional filters (like *audience*).

Turning Talks into Events

Get the code!

Get the code for this chapter (More info):

```
git checkout events
```

We forgot something: A list of talks is great especially if you can sort it by your preferences. But if a visitor decides he wants to actually go to see a talk he needs to know when it will take place.

We need a schedule and for this we need to store the information when a talk will happen.

Luckily the default type *Event* is based on reusable behaviors from the package plone.app.event.

In this chapter we will

- enable this behavior for talks
- display the date in the talkview and talklistview

First we enable the behavior `IEventBasic` for talks in `profiles/default/types/talk.xml`

```
<property name="behaviors">
  <element value="plone.app.dexterity.behaviors.metadata.IDublinCore"/>
  <element value="plone.app.content.interfaces.INameFromTitle"/>
  <element value="ploneconf.site.behavior.social.ISocial"/>
  <element value="ploneconf.site.interfaces.ITalk"/>
  <element value="plone.app.event.dx.behaviors.IEventBasic"/>
</property>
```

After we activate the behavior by hand or reinstalled the add-on we will now have some additional fields for `start` and `end`.

To display the new field we reuse a default event summary view as documented in http://ploneappevent.readthedocs.io/en/latest/development.html#reusing-the-event-summary-view-to-list-basic-event-information

Edit `browser/templates/talkview.pt`

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en"
      metal:use-macro="context/main_template/macros/master"
      i18n:domain="ploneconf.site">
<body>
    <metal:content-core fill-slot="content-core" tal:define="widgets view/w">

        <tal:eventsummary replace="structure context/@@event_summary"/>

        <p>
            <span tal:content="context/type_of_talk">
                Talk
            </span>
            suitable for
            <span tal:replace="structure widgets/audience/render">
                Audience
            </span>
        </p>

        <div tal:content="structure widgets/details/render">
            Details
        </div>

        <div class="newsImageContainer">
            <img tal:condition="python:getattr(context, 'image', None)"
                 tal:attributes="src string:${context/absolute_url}/@@images/image/
↪thumb" />
        </div>

        <div>
            <a class="email-link" tal:attributes="href string:mailto:${context/email}
↪">
                <strong tal:content="context/speaker">
                    Jane Doe
                </strong>
            </a>
            <div tal:content="structure widgets/speaker_biography/render">
                Biography
            </div>
        </div>

    </metal:content-core>
</body>
</html>
```

Similar to the field *room* the problem now appears that speakers submitting their talks should not be able to set a time and day for their talks. Sadly it is not easy to modify permissions of fields provided by behaviors (unless we write the bahvior ourselves). At least in this case we can take the easy way out since the field does not contain secret information: We will simply hide the fields from contributors using css and show them for reviewers. We will do so in chapter *Resources* when we add some css-files.

Modify `browser/static/ploneconf.css` and add:

```
body.userrole-contributor #formfield-form-widgets-IEventBasic-start,
body.userrole-contributor #formfield-form-widgets-IEventBasic-end > *,
body.userrole-contributor #formfield-form-widgets-IEventBasic-whole_day,
body.userrole-contributor #formfield-form-widgets-IEventBasic-open_end {
    display: none;
}
```

```
body.userrole-reviewer #formfield-form-widgets-IEventBasic-start,
body.userrole-reviewer #formfield-form-widgets-IEventBasic-end > *,
body.userrole-reviewer #formfield-form-widgets-IEventBasic-whole_day,
body.userrole-reviewer #formfield-form-widgets-IEventBasic-open_end {
    display: block;
}
```

You should also display the start-date of a talk in the talklist. Modify `browser/templates/talklistview.pt`

```
[...]
<td tal:content="python:talk['audience']">
    Advanced
</td>
<td class="pat-moment"
    data-pat-moment="format:calendar"
    tal:content="python:talk['start']">
    Time
</td>
<td tal:content="python:talk['room']">
    101
</td>
[...]
```

Exercise 1

Find out where `event_summary` comes from and describe how you could override it.

Solution

Use your editor or grep to search all zcml-files in the folder `packages` for the string `name="event_summary"`

```
$ grep -sirn --include \*.zcml 'name="event_summary"' ./packages
./packages/plone/app/event/browser/configure.zcml:66:        name="event_summary"
./packages/plone/app/event/browser/configure.zcml:75:        name="event_summary"
```

The relevant registration is:

```
<browser:page
    for="plone.event.interfaces.IEvent"
    name="event_summary"
    class=".event_summary.EventSummaryView"
    template="event_summary.pt"
    permission="zope2.View"
    layer="..interfaces.IBrowserLayer"
    />
```

So there is a class `plone.app.event.browser.event_summary.EventSummaryView` and a template `event_summary.pt` that could be overridden with `z3c.jbot` by copying it as `plone.app.event.browser.event_summary.pt` in `browser/overrides`.

Exercise 2

Find out where the event behavior is defined and which fields it offers.

Solution

The id with which the behavior is registered in `Talk.xml` is a Python path. So `plone.app.event.dx.behaviors.IEventBasic` can be found in `packages/plone.app.event/plone/app/event/dx/behaviors.py`

```
class IEventBasic(model.Schema, IDXEvent):
    """ Basic event schema.
    """
    start = schema.Datetime(
        title=_(
            u'label_event_start',
            default=u'Event Starts'
        ),
        description=_(
            u'help_event_start',
            default=u'Date and Time, when the event begins.'
        ),
        required=True,
        defaultFactory=default_start
    )

    end = schema.Datetime(
        title=_(
            u'label_event_end',
            default=u'Event Ends'
        ),
        description=_(
            u'help_event_end',
            default=u'Date and Time, when the event ends.'
        ),
        required=True,
        defaultFactory=default_end
    )

    whole_day = schema.Bool(
        title=_(
            u'label_event_whole_day',
            default=u'Whole Day'
        ),
        description=_(
            u'help_event_whole_day',
            default=u'Event lasts whole day.'
        ),
        required=False,
        default=False
    )

    open_end = schema.Bool(
        title=_(
            u'label_event_open_end',
            default=u'Open End'
        ),
        description=_(
            u'help_event_open_end',
            default=u"This event is open ended."
        ),
        required=False,
        default=False
    )
```

Note how it uses `defaultFactory` to set an initial value.

User Generated Content

Get the code!

Get the code for this chapter (More info):

```
git checkout user_generated_content
```

How do prospective speakers submit talks? We let them register on the site and grant right to create talks. For this we go back to changing the site through-the-web.

In this chapter we:

- allow self-registration
- constrain types on the talk folder
- grant local roles
- create a custom workflow for talks

Self-registration

- Go to the Security control panel at http://localhost:8080/Plone/@@security-controlpanel and Enable self-registration
- Leave "Enable User Folders" off unless you want a community site.

Constrain types

- On the talk folder select Restrictions… from the *Add new* menu. Only allow to add talks.

Grant local roles

- Go to *Sharing* and grant the role *Can add* to the group logged-in users. Now every user can add content in this folder (and only this folder).

Now all logged-in users can create and submit talks in this folder with the permission of the default workflow.

A custom workflow for talks

We still need to fix a problem: Authenticated users can see all talks, even the ones of other users in the private state. Since we don't want this we will create a modified workflow for talks. The new workflow will only let them see and edit talks they created themselves and not the ones of other users.

- Go to the *ZMI* → *portal_workflow*

- See how talks have the same workflow as most content, namely *(Default)*
- Go to the tab *Contents*, check the box next to *simple_publication_workflow*, click *copy* and *paste*.
- Rename the new workflow from *copy_of_simple_publication_workflow* to *talks_workflow*.
- Edit the workflow by clicking on it: Change the Title to *Talks Workflow*.
- Click on the tab *States* and click on *private* to edit this state. In the next view select the tab *Permissions*.
- Find the table column for the role *Contributor* and remove the permissions for *Access contents information* and *View*. Note that the *Owner* (i.e. the Creator) still has some permissions.
- Do the same for the state *pending*
- Go back to `portal_workflow` and set the new workflow `talks_workflow` for talks. Click `Change` and then `Update security settings`.

Note: The add-on plone.app.workflowmanager provides a much nicer user-interface for this. The problem is you need a big screen for it and it can be pretty confusing as well.

Done.

Move the changes to the file system

We don't want to do these steps for every new conference by hand so we move the changes into our package.

Import/Export the Workflow

- export the GenericSetup step *Workflow Tool* in http://localhost:8080/Plone/portal_setup/manage_exportSteps.
- drop the file `workflows.xml` into `profiles/default` an clean out everything that is not related to talks.

```
<?xml version="1.0"?>
<object name="portal_workflow" meta_type="Plone Workflow Tool">
 <object name="talks_workflow" meta_type="Workflow"/>
 <bindings>
  <type type_id="talk">
   <bound-workflow workflow_id="talks_workflow"/>
  </type>
 </bindings>
</object>
```

- drop `workflows/talks_workflow/definition.xml` in `profiles/default/workflows/talks_workflow/` The other files are just definitions of the default-workflows and we only want things in our package that changes Plone.

Enable self-registration

To enable self-registration you need to change the global setting that controls this option. Most global setting are stored in the registry. You can modify it by adding following to `profiles/default/registry.xml`:

```
<record name="plone.enable_self_reg">
  <value>True</value>
</record>
```

Grant local roles

Since the granting of local roles applies only to a certain folder in the site we would not always write code for it but do it by hand. But for testability and repeatability (there is a conference every year!) we should create the initial content structure automatically.

So let's make sure some initial content is created and configured on installing the package.

To run arbitrary code during the installation of a package we use a post_handler

Our package already has such an method registered in `configure.zcml`. It will be automatically run when (re-)installing the add-on.

```
<genericsetup:registerProfile
    name="default"
    title="ploneconf.site"
    directory="profiles/default"
    description="Installs the ploneconf.site add-on."
    provides="Products.GenericSetup.interfaces.EXTENSION"
    post_handler=".setuphandlers.post_install"
    />
```

This makes sure the method `post_install()` in `setuphandlers.py` is executed after the installation. The method already exists doing nothing. You need to extend it to do what we want.

```
# -*- coding: utf-8 -*-
from plone import api
from Products.CMFPlone.interfaces import constrains
from Products.CMFPlone.interfaces import INonInstallable
from zope.interface import implementer

import logging

logger = logging.getLogger(__name__)
PROFILE_ID = 'profile-ploneconf.site:default'


@implementer(INonInstallable)
class HiddenProfiles(object):

    def getNonInstallableProfiles(self):
        """Hide uninstall profile from site-creation and quickinstaller"""
        return [
            'ploneconf.site:uninstall',
        ]


def post_install(context):
    """Post install script"""
    # Do something at the end of the installation of this package.
    portal = api.portal.get()
    set_up_content(portal)


def set_up_content(portal):
    """Create and configure some initial content.
    Part of this code is taken from upgrades.py
    """
    # Create a folder 'The event' if needed
    if 'the-event' not in portal:
        event_folder = api.content.create(
            container=portal,
            type='Folder',
            id='the-event',
            title=u'The event')
    else:
        event_folder = portal['the-event']

    # Create folder 'Talks' inside 'The event' if needed
    if 'talks' not in event_folder:
        talks_folder = api.content.create(
            container=event_folder,
            type='Folder',
            id='talks',
            title=u'Talks')
    else:
        talks_folder = event_folder['talks']

    # Allow logged-in users to create content
    api.group.grant_roles(
        groupname='AuthenticatedUsers',
        roles=['Contributor'],
        obj=talks_folder)

    # Constrain addable types to talk
    behavior = constrains.ISelectableConstrainTypes(talks_folder)
    behavior.setConstrainTypesMode(constrains.ENABLED)
    behavior.setLocallyAllowedTypes(['talk'])
    behavior.setImmediatelyAddableTypes(['talk'])
    logger.info('Added and configured {0}'.format(talks_folder.absolute_url()))


def uninstall(context):
    """Uninstall script"""
    # Do something at the end of the uninstallation of this package.
```

Once we reinstall our package a folder `talks` is created with the appropriate local roles and constraints.

We wrote similar code to create the folder *The Event* in *Upgrade steps*. We need it to make sure a sane structure gets created when we create a new site by hand or in tests.

You would usually create a list of dictionaries containing the type, parent and title plus optionally layout, workflow state etc. to create an initial structure. In some projects it could also make sense to have a separate profile besides `default` which might be called `demo` or `content` that creates an initial structure and maybe another `testing` that creates dummy content (talks, speakers etc) for tests.

Exercise 1

Create a profile `content` that runs its own post_handler in `setuphandlers.py`.

Solution

Register the profile and the upgrade step in `configure.zcml`

```
<genericsetup:registerProfile
    name="content"
    title="PloneConf Site initial content"
    directory="profiles/content"
    description="Extension profile for PloneConf Talk to add initial content"
    provides="Products.GenericSetup.interfaces.EXTENSION"
    post_handler=".setuphandlers.post_content"
    />
```

Also add a `profiles/content/metadata.xml` so the default profile gets automatically installed when installing the content profile.

```
<metadata>
  <version>1000</version>
  <dependencies>
    <dependency>profile-ploneconf.site:default</dependency>
  </dependencies>
</metadata>
```

Add the structure you wish to create as a list of dictionaries in `setuphandlers.py`:

```
STRUCTURE = [
    {
        'type': 'Folder',
        'title': u'The Event',
        'id': 'the-event',
        'description': u'Plone Conference 2020',
        'default_page': 'frontpage-for-the-event',
        'state': 'published',
        'children': [{
            'type': 'Document',
            'title': u'Frontpage for the-event',
            'id': 'frontpage-for-the-event',
            'state': 'published',
            },
            {
            'type': 'Folder',
            'title': u'Talks',
            'id': 'talks',
            'layout': 'talklistview',
            'state': 'published',
            },
            {
            'type': 'Folder',
            'title': u'Training',
            'id': 'training',
            'state': 'published',
            },
            {
            'type': 'Folder',
            'title': u'Sprint',
            'id': 'sprint',
            'state': 'published',
            },
        ]
    },
    {
        'type': 'Folder',
        'title': u'Talks',
        'id': 'talks',
        'description': u'Submit your talks here!',
        'state': 'published',
        'layout': '@@talklistview',
        'allowed_types': ['talk'],
        'local_roles': [{
            'group': 'AuthenticatedUsers',
            'roles': ['Contributor']
        }],
    },
    {
        'type': 'Folder',
        'title': u'News',
        'id': 'news',
        'description': u'News about the Plone Conference',
        'state': 'published',
        'children': [{
            'type': 'News Item',
            'title': u'Submit your talks!',
            'id': 'submit-your-talks',
            'description': u'Task submission is open',
            'state': 'published', }
        ],
    },
    {
        'type': 'Folder',
        'title': u'Events',
        'id': 'events',
        'description': u'Dates to keep in mind',
        'state': 'published',
    },
]
```

Add the method `content()` to `setuphandlers.py`. We pointed to that when registering the import step. And add some fancy logic to create the content from `STRUCTURE`.

```
from zope.lifecycleevent import modified


def post_content(context):
    portal = api.portal.get()
    for item in STRUCTURE:
        _create_content(item, portal)


def _create_content(item_dict, container, force=False):
    if not force and container.get(item_dict['id'], None) is not None:
        return

    # Extract info that can't be passed to api.content.create
    layout = item_dict.pop('layout', None)
    default_page = item_dict.pop('default_page', None)
    allowed_types = item_dict.pop('allowed_types', None)
    local_roles = item_dict.pop('local_roles', [])
    children = item_dict.pop('children', [])
    state = item_dict.pop('state', None)

    new = api.content.create(
        container=container,
        safe_id=True,
        **item_dict
    )
    logger.info('Created {0} at {1}'.format(new.portal_type, new.absolute_url()))

    if layout is not None:
        new.setLayout(layout)
    if default_page is not None:
        new.setDefaultPage(default_page)
    if allowed_types is not None:
        _constrain(new, allowed_types)
    for local_role in local_roles:
        api.group.grant_roles(
            groupname=local_role['group'],
            roles=local_role['roles'],
            obj=new)
    if state is not None:
        api.content.transition(new, to_state=state)

    modified(new)
    # call recursively for children
    for subitem in children:
        _create_content(subitem, new)


def _constrain(context, allowed_types):
    behavior = constrains.ISelectableConstrainTypes(context)
    behavior.setConstrainTypesMode(constrains.ENABLED)
    behavior.setLocallyAllowedTypes(allowed_types)
    behavior.setImmediatelyAddableTypes(allowed_types)
```

A huge benefit of this implementation is that you can add any object-attribute as a new item to `item_dict`. `plone.api.content.create()` will then set these on the new objects. This way you can also populate fields like `text` (using `plone.app.textfield.RichTextValue`) or `image` (using `plone.namedfile.file.NamedBlobImage`).

Resources

Get the code!

Get the code for this chapter (More info):

```
git checkout resources
```

We have not yet talked about CSS and Javascript. At the moment these are considered static resources.

You can declare and access static resources with special urls. The *configure.zcml* of our package already has a declaration for a resource-folder `static`.

```
<plone:static
    name="ploneconf.site"
    type="plone"
    directory="static"
    />
```

All files we put in the `static` folder can be accessed via the url http://localhost:8080/Plone/++plone++ploneconf.site/the_real_filename.css

Another feature of this folder ist that the resouces you put in there are editable and overrideable in the browser using the overrides-tab of the resource registry.

Let's create a file `ploneconf.css` in the `static` folder with some CSS:

```
header #portal-header #portal-searchbox .searchSection {
    display: none;
}

body.userrole-contributor #formfield-form-widgets-IEventBasic-start,
body.userrole-contributor #formfield-form-widgets-IEventBasic-end > *,
body.userrole-contributor #formfield-form-widgets-IEventBasic-whole_day,
body.userrole-contributor #formfield-form-widgets-IEventBasic-open_end {
    display: none;
}

body.userrole-reviewer #formfield-form-widgets-IEventBasic-start,
body.userrole-reviewer #formfield-form-widgets-IEventBasic-end > *,
body.userrole-reviewer #formfield-form-widgets-IEventBasic-whole_day,
body.userrole-reviewer #formfield-form-widgets-IEventBasic-open_end {
    display: block;
}
```

The css is not very exciting. It hides the *only in current section* below the search-box (we could also overwrite the viewlet, but ...). It also hides the event-fields we added in *Turning Talks into Events* from people submitting their talks. For exiting css you take the training *Mastering Plone Theming*.

If we now access http://localhost:8080/Plone/++plone++ploneconf.site/ploneconf.css we see our css-file.

Also add a `ploneconf.js` in the same folder but leave it empty for now. You could add some JavaScript to that file later.

How do our JavaScript and CSS files get used when visiting the page? So far the new files are accessible in the browser but we want Plone to use them every time we access the page. Adding them directly into the HTML is not a good solution, having many CSS and JS files slows down the page loading.

For this we need to register a *bundle* that contains these files. Plone will then make sure that all files that are part of this bundle are also deployed. We need to register our resources with GenericSetup.

Open the file `profiles/default/registry.xml` and add the following:

```
<!-- the plonconf bundle -->
<records prefix="plone.bundles/ploneconf-bundle"
         interface='Products.CMFPlone.interfaces.IBundleRegistry'>
  <value key="resources">
    <element>ploneconf-main</element>
  </value>
  <value key="enabled">True</value>
  <value key="compile">True</value>
  <value key="csscompilation">++plone++ploneconf.site/ploneconf.css</value>
  <value key="jscompilation">++plone++ploneconf.site/ploneconf.js</value>
  <value key="last_compilation"></value>
</records>
```

The resources that are part of the registered bundle will now be deployed with every request.

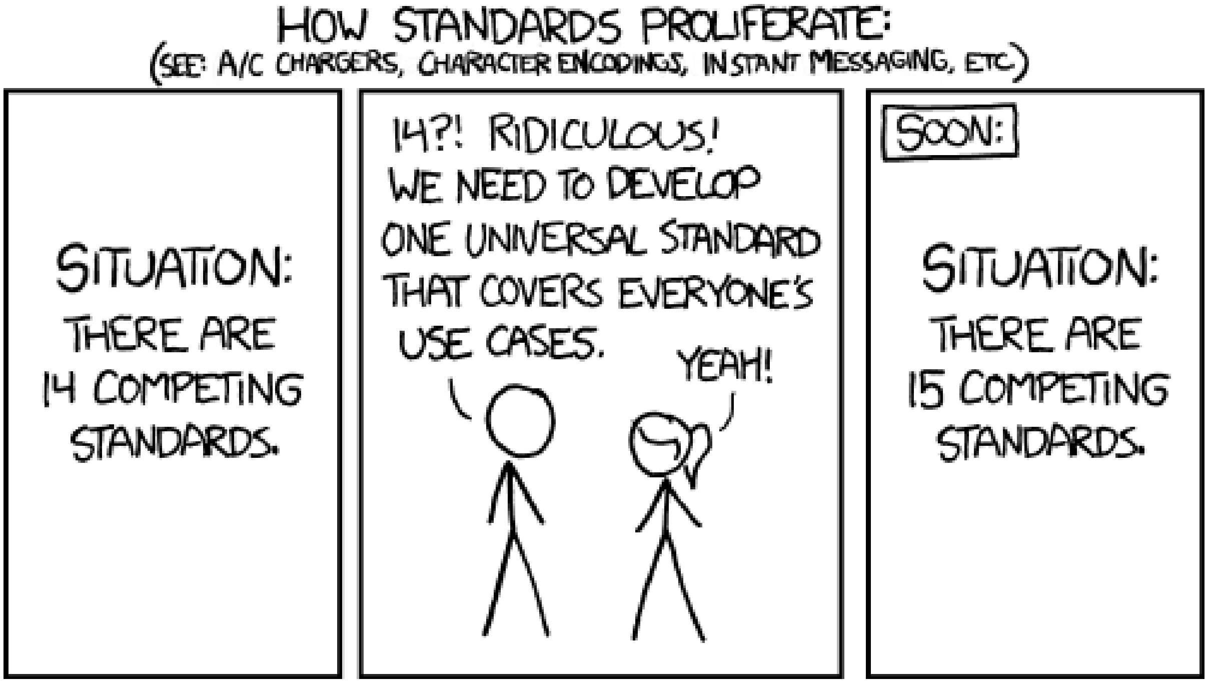
For more infos please see http://docs.plone.org/adapt-and-extend/theming/resourceregistry.html or https://training.plone.org/5/theming/adv-resource-registry.html.

Using Third-Party Behaviors

Warning: Skip this since collective.behavior.banner is not yet compatible with Plone 5.

Add teaser with collective.behavior.banner

There are a lot of add-ons in Plone for sliders/banners/teasers. We thought there should be a better one and created `collective.behavior.banner`.


HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC)
SITUATION: THERE ARE 14 COMPETING STANDARDS.
14?! RIDICULOUS! WE NEED TO DEVELOP ONE UNIVERSAL STANDARD THAT COVERS EVERYONE'S USE CASES.
YEAH!
SOON:
SITUATION: THERE ARE 15 COMPETING STANDARDS.

Like many add-ons it has not yet been released on pypi but only exists as code on github.

The training buildout has a section `[sources]` that tells buildout to download a specific add-on not from pypi but from some code repository (usually github):

```
[sources]
collective.behavior.banner = git https://github.com/collective/collective.behavior.
↪banner.git pushurl=git@github.com:collective/collective.behavior.banner.git␣
↪rev=af2dc1f21b23270e4b8583cf04eb8e962ade4c4d
```

Pinning the revision saves us from being surprised by changes in the code we might not want.

After adding the source, we need to add the egg to buildout:

```
eggs =
    Plone
    ...
    collective.behavior.banner
    ...
```

And rerun `./bin/buildout`

- Install the add-on
- Create a new dexterity content type `Banner` with **only** the behavior `Banner` enabled.
- Create a folder called `banners`
- Add two banners into that folder using images taken from lorempixel.com
- Add the Behavior `Slider` to the default content type `Page (Document)`
- Edit the front-page and link to the new banners.

Dexterity Types III: Python

Get the code!

Get the code for this chapter (More info):

```
git checkout dexterity_3
```

Without sponsors, a conference would be hard to finance! Plus it is a good opportunity for Plone companies to advertise their services. But sponsors want to be displayed in a nice way according to the size of their sponsorship.

In this part we will:

- create the content type *sponsor* that has a Python schema,
- create a viewlet that shows the sponsor logos sorted by sponsoring level.

The topics we cover are:

- Python schema for Dexterity
- schema hint and directives
- field permissions
- image scales
- caching

The Python schema

First we create the schema for the new type. Instead of XML, we use Python this time. In chapter *Return to Dexterity: Moving contenttypes into Code* you already created a folder `content` with an empty `__init__.py` in it. We don't need to register that folder in `configure.zcml` since we don't need a `content/configure.zcml` (at least not yet).

Now add a new file `content/sponsor.py`.

```python
# -*- coding: utf-8 -*-
from plone.app.textfield import RichText
from plone.autoform import directives
from plone.namedfile import field as namedfile
from plone.supermodel import model
from plone.supermodel.directives import fieldset
from ploneconf.site import _
from z3c.form.browser.radio import RadioFieldWidget
from zope import schema
from zope.schema.vocabulary import SimpleTerm
from zope.schema.vocabulary import SimpleVocabulary


LevelVocabulary = SimpleVocabulary(
    [SimpleTerm(value=u'platinum', title=_(u'Platinum Sponsor')),
     SimpleTerm(value=u'gold', title=_(u'Gold Sponsor')),
     SimpleTerm(value=u'silver', title=_(u'Silver Sponsor')),
     SimpleTerm(value=u'bronze', title=_(u'Bronze Sponsor'))]
    )


class ISponsor(model.Schema):
    """Dexterity Schema for Sponsors
    """

    directives.widget(level=RadioFieldWidget)
    level = schema.Choice(
        title=_(u'Sponsoring Level'),
        vocabulary=LevelVocabulary,
        required=True
    )

    text = RichText(
        title=_(u'Text'),
        required=False
    )

    url = schema.URI(
        title=_(u'Link'),
        required=False
    )

    fieldset('Images', fields=['logo', 'advertisement'])
    logo = namedfile.NamedBlobImage(
        title=_(u'Logo'),
        required=False,
    )

    advertisement = namedfile.NamedBlobImage(
        title=_(u'Advertisement (Gold-sponsors and above)'),
        required=False,
    )

    directives.read_permission(notes='cmf.ManagePortal')
    directives.write_permission(notes='cmf.ManagePortal')
    notes = RichText(
        title=_(u'Secret Notes (only for site-admins)'),
        required=False
    )
```

Some things are notable here:

- The fields in the schema are mostly from `zope.schema`. A reference of available fields is at http://docs.plone.org/external/plone.app.dexterity/docs/reference/fields.html
- In `directives.widget(level=RadioFieldWidget)` we change the default widget for a Choice field from a dropdown to radio-boxes. An incomplete reference of available widgets is at http://docs.plone.org/external/plone.app.dexterity/docs/reference/widgets.html
- `LevelVocabulary` is used to create the options used in the field `level`. This way we could easily translate the displayed value.
- `fieldset('Images',fields=['logo','advertisement'])` moves the two image fields to another tab.
- `directives.read_permission(...)` sets the read and write permission for the field `notes` to users who can add new members. Usually this permission is only granted to Site Administrators and Managers. We use it to store information that should not be publicly visible. Please note that `obj.notes` is still accessible in templates and Python. Only using the widget (like we do in the view later) checks for the permission.
- We use no grok here.

See also:

- All available Fields
- Schema-driven types with Dexterity
- Form schema hints and directives

The FTI

Second we create the FTI for the new type in `profiles/default/types/sponsor.xml`

```xml
<?xml version="1.0"?>
<object name="sponsor" meta_type="Dexterity FTI" i18n:domain="plone"
   xmlns:i18n="http://xml.zope.org/namespaces/i18n">
 <property name="title" i18n:translate="">Sponsor</property>
 <property name="description" i18n:translate=""></property>
 <property name="icon_expr">string:${portal_url}/document_icon.png</property>
 <property name="factory">sponsor</property>
 <property name="add_view_expr">string:${folder_url}/++add++sponsor</property>
 <property name="link_target"></property>
 <property name="immediate_view">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">view</property>
 <property name="view_methods">
  <element value="view"/>
 </property>
 <property name="default_view_fallback">False</property>
 <property name="add_permission">cmf.AddPortalContent</property>
 <property name="klass">plone.dexterity.content.Container</property>
 <property name="behaviors">
  <element value="plone.app.dexterity.behaviors.metadata.IDublinCore"/>
  <element value="plone.app.content.interfaces.INameFromTitle"/>
 </property>
 <property name="schema">ploneconf.site.content.sponsor.ISponsor</property>
 <property name="model_source"></property>
 <property name="model_file"></property>
 <property name="schema_policy">dexterity</property>
 <alias from="(Default)" to="(dynamic view)"/>
 <alias from="edit" to="@@edit"/>
 <alias from="sharing" to="@@sharing"/>
 <alias from="view" to="(selected layout)"/>
 <action title="View" action_id="view" category="object" condition_expr=""
    description="" icon_expr="" link_target="" url_expr="string:${object_url}"
    visible="True">
  <permission value="View"/>
 </action>
 <action title="Edit" action_id="edit" category="object" condition_expr=""
    description="" icon_expr="" link_target=""
    url_expr="string:${object_url}/edit" visible="True">
  <permission value="Modify portal content"/>
 </action>
</object>
```

Then we register the FTI in `profiles/default/types.xml`

```
<?xml version="1.0"?>
<object name="portal_types" meta_type="Plone Types Tool">
 <property name="title">Controls the available contenttypes in your portal</property>
 <object name="talk" meta_type="Dexterity FTI"/>
 <object name="sponsor" meta_type="Dexterity FTI"/>
 <!-- -*- more types can be added here -*- -->
</object>
```

After reinstalling our package we can create the new type.

Exercise 1

Sponsors are containers but they don’t need to be. Turn them into items by changing their class to `plone.dexterity.content.Item`.

Solution

Simply modify the property `klass` in the FTI and reinstall.

```
<property name="klass">plone.dexterity.content.Item</property>
```

The view

We use the default view provided by dexterity for testing since we will only display the sponsors in a viewlet and not in their own page.

But we could tweak the default view with some CSS to make it less ugly. Add the following to `resources/ploneconf.css`:

```
.template-view.portaltype-sponsor .named-image-widget img {
    width: 100%;
    height: auto;
}

.template-view.portaltype-sponsor fieldset#folder-listing {
    display: none;
}
```

Note: If we really want a custom view for sponsors it could look like this.

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en"
      metal:use-macro="context/main_template/macros/master"
      i18n:domain="ploneconf.site">
<body>
  <metal:content-core fill-slot="content-core">
    <h3 tal:content="structure view/w/level/render">
      Level
    </h3>

    <div tal:content="structure view/w/text/render">
      Text
    </div>

    <div class="newsImageContainer">
      <a tal:attributes="href context/url">
        <img tal:condition="python:getattr(context, 'logo', None)"
             tal:attributes="src string:${context/absolute_url}/@@images/logo/preview
↪" />
      </a>
    </div>

    <div>
      <a tal:attributes="href context/url">
        Website
      </a>

      <img tal:condition="python:getattr(context, 'advertisement', None)"
           tal:attributes="src string:${context/absolute_url}/@@images/advertisement/
↪preview" />

      <div tal:condition="python: 'notes' in view.w"
           tal:content="structure view/w/notes/render">
        Notes
      </div>

    </div>
  </metal:content-core>
</body>
</html>
```

Note how we handle the field with special permissions: `tal:condition="python: 'notes' in view.w"` checks if the convenience-dictionary `w` (provided by the base class `DefaultView`) holds the widget for the field `notes`. If the current user does not have the permission `cmf.ManagePortal` it will be omitted from the dictionary and get an error since `notes` would not be a key in `w`. By first checking if it's missing we work around that.

The viewlet

Instead of writing a view you will have to display the sponsors at the bottom of the website in a viewlet.

Register the viewlet in `browser/configure.zcml`

```xml
<browser:viewlet
    name="sponsorsviewlet"
    manager="plone.app.layout.viewlets.interfaces.IPortalFooter"
    for="*"
    layer="..interfaces.IPloneconfSiteLayer"
    class=".viewlets.SponsorsViewlet"
    template="templates/sponsors_viewlet.pt"
    permission="zope2.View"
    />
```

Add the viewlet class in `browser/viewlets.py`

```python
# -*- coding: utf-8 -*-
from collections import OrderedDict
from plone import api
from plone.app.layout.viewlets.common import ViewletBase
from plone.memoize import ram
from ploneconf.site.behaviors.social import ISocial
from ploneconf.site.content.sponsor import LevelVocabulary
from random import shuffle
from time import time


class SocialViewlet(ViewletBase):

    def lanyrd_link(self):
        adapted = ISocial(self.context)
        return adapted.lanyrd


class SponsorsViewlet(ViewletBase):

    @ram.cache(lambda *args: time() // (60 * 60))
    def _sponsors(self):
        results = []
        for brain in api.content.find(portal_type='sponsor'):
            obj = brain.getObject()
            scales = api.content.get_view(
                name='images',
                context=obj,
                request=self.request)
            scale = scales.scale(
                'logo',
                width=200,
                height=80,
                direction='down')
            tag = scale.tag() if scale else None
            if not tag:
                # only display sponsors with a logo
                continue
            results.append({
                'title': obj.title,
                'description': obj.description,
                'tag': tag,
                'url': obj.url or obj.absolute_url(),
                'level': obj.level
            })
        return results

    def sponsors(self):
        sponsors = self._sponsors()
        if not sponsors:
            return
        results = OrderedDict()
        levels = [i.value for i in LevelVocabulary]
        for level in levels:
            level_sponsors = []
            for sponsor in sponsors:
                if level == sponsor['level']:
                    level_sponsors.append(sponsor)
            if not level_sponsors:
                continue
            shuffle(level_sponsors)
            results[level] = level_sponsors
        return results
```

- `_sponsors()` returns a list of dictionaries containing all necessary info about sponsors.
- We create the complete img tag using a custom scale (200x80) using the view `images` from `plone.namedfile`. This actually scales the logos and saves them as new blobs.
- In `sponsors()` we return an ordered dictionary of randomized lists of dicts (containing the information on sponsors). The order is by sponsor-level since we want the platinum-sponsors on top and the bronze-sponsors at the bottom. The randomization is for fairness among equal sponsors.

`_sponsors()` is cached for an hour using plone.memoize. This way we don't need to keep all sponsor objects in memory all the time. But we'd have to wait for up to an hour until changes will be visible.

Instead we should cache until one of the sponsors is modified by using a callable `_sponsors_cachekey()` that returns a number that changes when a sponsor is modified.

```
...
def _sponsors_cachekey(method, self):
    brains = api.content.find(portal_type='sponsor')
    cachekey = sum([int(i.modified) for i in brains])
    return cachekey

@ram.cache(_sponsors_cachekey)
def _sponsors(self):
    catalog = api.portal.get_tool('portal_catalog')
...
```

See also:

- Guide to Caching
- Cache decorators
- Image Scaling

The template for the viewlet

Add the template `browser/templates/sponsors_viewlet.pt`

```
<div metal:define-macro="portal_sponsorbox"
     i18n:domain="ploneconf.site">
```

```
    <div id="portal-sponsorbox" class="container"
         tal:define="sponsors view/sponsors;"
         tal:condition="sponsors">
        <div class="row">
            <h2>We  our sponsors</h2>
        </div>
        <div tal:repeat="level sponsors"
             tal:attributes="id python:'level-' + level"
             class="row">
            <h3 tal:content="python: level.capitalize()">
                Gold
            </h3>
            <tal:images tal:define="items python:sponsors[level];"
                        tal:repeat="item items">
                <div class="sponsor">
                    <a href=""
                       tal:attributes="href python:item['url'];
                                       title python:item['title'];">
                        <img tal:replace="structure python:item['tag']" />
                    </a>
                </div>
            </tal:images>
        </div>
    </div>
</div>
```

You can now add some CSS in `browser/static/ploneconf.css` to make it look OK.

```
.sponsor {
    display: inline-block;
    margin: 0 1em 1em 0;
}

.sponsor:hover {
    box-shadow: 0 0 8px #000;
    -moz-box-shadow: 0 0 8px #000;
    -webkit-box-shadow: 0 0 8px #000;
}
```

Exercise 2

Turn the content type Speaker from *Exercise 2 of the first chapter on dexterity* into a Python-based type.

When we're done, it should have the following fields:

- title
- email
- homepage
- biography
- company
- twitter_name
- irc_name

- image

Do *not* use the `IBasic` or `IDublinCore` behavior to add title and description. Instead add your own field `title` and give it the title *Name*.

Solution

```
# -*- coding: utf-8 -*-
from plone.app.textfield import RichText
from plone.app.vocabularies.catalog import CatalogSource
from plone.autoform import directives
from plone.namedfile import field as namedfile
from plone.supermodel import model
from ploneconf.site import _
from z3c.relationfield.schema import RelationChoice
from z3c.relationfield.schema import RelationList
from zope import schema


class ISpeaker(model.Schema):
    """Dexterity-Schema for Speaker
    """

    first_name = schema.TextLine(
        title=_(u'First Name'),
    )

    last_name = schema.TextLine(
        title=_(u'Last Name'),
    )

    email = schema.TextLine(
        title=_(u'E-Mail'),
        required=False,
    )

    homepage = schema.URI(
        title=_(u'Homepage'),
        required=False,
    )

    biography = RichText(
        title=_(u'Biography'),
        required=False,
    )

    company = schema.TextLine(
        title=_(u'Company'),
        required=False,
    )

    twitter_name = schema.TextLine(
        title=_(u'Twitter-Name'),
        required=False,
    )

    irc_name = schema.TextLine(
        title=_(u'IRC-Name'),
        required=False,
    )

    image = namedfile.NamedBlobImage(
        title=_(u'Image'),
        required=False,
    )
```

Register the type in `profiles/default/types.xml`

```
<?xml version="1.0"?>
<object name="portal_types" meta_type="Plone Types Tool">
 <property name="title">Controls the available contenttypes in your portal</property>
 <object name="talk" meta_type="Dexterity FTI"/>
 <object name="sponsor" meta_type="Dexterity FTI"/>
 <object name="speaker" meta_type="Dexterity FTI"/>
 <!-- -*- more types can be added here -*- -->
</object>
```

The FTI goes in `profiles/default/types/speaker.xml`. Again we use `Item` as the base-class:

```
<?xml version="1.0"?>
<object name="speaker" meta_type="Dexterity FTI" i18n:domain="plone"
   xmlns:i18n="http://xml.zope.org/namespaces/i18n">
 <property name="title" i18n:translate="">Speaker</property>
 <property name="description" i18n:translate=""></property>
 <property name="icon_expr">string:${portal_url}/document_icon.png</property>
 <property name="factory">speaker</property>
 <property name="add_view_expr">string:${folder_url}/++add++speaker</property>
 <property name="link_target"></property>
 <property name="immediate_view">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">view</property>
 <property name="view_methods">
  <element value="view"/>
 </property>
 <property name="default_view_fallback">False</property>
 <property name="add_permission">cmf.AddPortalContent</property>
 <property name="klass">plone.dexterity.content.Item</property>
 <property name="behaviors">
  <element value="plone.app.dexterity.behaviors.metadata.IBasic"/>
  <element value="plone.app.content.interfaces.INameFromTitle"/>
 </property>
 <property name="schema">ploneconf.site.content.speaker.ISpeaker</property>
 <property name="model_source"></property>
 <property name="model_file"></property>
 <property name="schema_policy">dexterity</property>
 <alias from="(Default)" to="(dynamic view)"/>
 <alias from="edit" to="@@edit"/>
 <alias from="sharing" to="@@sharing"/>
 <alias from="view" to="(selected layout)"/>
 <action title="View" action_id="view" category="object" condition_expr=""
    description="" icon_expr="" link_target="" url_expr="string:${object_url}"
    visible="True">
  <permission value="View"/>
 </action>
 <action title="Edit" action_id="edit" category="object" condition_expr=""
    description="" icon_expr="" link_target=""
    url_expr="string:${object_url}/edit" visible="True">
  <permission value="Modify portal content"/>
 </action>
</object>
```

After reinstalling the package the new type is usable.

Exercise 3

This is more of a Python exercise. The gold- and bronze sponsors should also have a bigger logo than the others. Give the sponsors the following logo-sizes without using CSS.

- Platinum: 500x200
- Gold: 350x150
- Silver: 200x80
- Bronze: 150x60

Solution

```python
# -*- coding: utf-8 -*-
from collections import OrderedDict
from plone import api
from plone.app.layout.viewlets.common import ViewletBase
from plone.memoize import ram
from ploneconf.site.behaviors.social import ISocial
from ploneconf.site.content.sponsor import LevelVocabulary
from random import shuffle

LEVEL_SIZE_MAPPING = {
    'platinum': (500, 200),
    'gold': (350, 150),
    'silver': (200, 80),
    'bronze': (150, 60),
}


class SocialViewlet(ViewletBase):

    def lanyrd_link(self):
        adapted = ISocial(self.context)
        return adapted.lanyrd


class SponsorsViewlet(ViewletBase):

    def _sponsors_cachekey(method, self):
        brains = api.content.find(portal_type='sponsor')
        cachekey = sum([int(i.modified) for i in brains])
        return cachekey

    @ram.cache(_sponsors_cachekey)
    def _sponsors(self):
        results = []
        for brain in api.content.find(portal_type='sponsor'):
            obj = brain.getObject()
            scales = api.content.get_view(
                name='images',
                context=obj,
                request=self.request)
            width, height = LEVEL_SIZE_MAPPING[obj.level]
            scale = scales.scale(
                'logo',
                width=width,
                height=height,
                direction='down')
            tag = scale.tag() if scale else None
            if not tag:
                # only display sponsors with a logo
                continue
            results.append({
                'title': obj.title,
                'description': obj.description,
                'tag': tag,
                'url': obj.url or obj.absolute_url(),
                'level': obj.level
            })
        return results

    def sponsors(self):
        sponsors = self._sponsors()
        if not sponsors:
            return
        results = OrderedDict()
        levels = [i.value for i in LevelVocabulary]
        for level in levels:
            level_sponsors = []
            for sponsor in sponsors:
                if level == sponsor['level']:
                    level_sponsors.append(sponsor)
            if not level_sponsors:
                continue
            shuffle(level_sponsors)
            results[level] = level_sponsors
        return results
```

Relations

You can model relationships between content items by placing them in a hierarchy (a folder *speakers* containing the (folderish) speakers and within each speaker the talks) or by linking them to each other in Richtext-Fields. But where would you store a talk that two speakers give together?

Relations allow developers to model relationships between objects without a links or a hierarchy. The behavior `plone.app.relationfield.behavior.IRelatedItems` provides the field *Related Items* in the tab *Categorization*. That field simply says `a` is somehow related to `b`.

By using custom relations you can model your data in a much more meaningful way.

Creating relations in a schema

Relate to one item only.

```
from plone.app.vocabularies.catalog import CatalogSource
from z3c.relationfield.schema import RelationChoice
from z3c.relationfield.schema import RelationList

evil_mastermind = RelationChoice(
    title=_(u'The Evil Masterimind'),
    vocabulary='plone.app.vocabularies.Catalog',
    required=False,
)
```

Relate to multiple items.

```
from z3c.relationfield.schema import RelationChoice
from z3c.relationfield.schema import RelationList

minions = RelationList(
    title=_(u'Minions'),
    default=[],
    value_type=RelationChoice(
        vocabulary='plone.app.vocabularies.Catalog',
    )
    required=False,
)
```

We can see that the code for the behavior IRelatedItems does exactly the same.

Instead of using a named vocabulary we can also use `source`:

```
from plone.app.vocabularies.catalog import CatalogSource
from z3c.relationfield.schema import RelationChoice
from z3c.relationfield.schema import RelationList

minions = RelationList(
    title=_(u'Talks by this speaker'),
    value_type=RelationChoice(
        title=_(u'Talks'),
        source=CatalogSource(portal_type=['one_eyed_minion', 'minion'])),
    required=False,
)
```

To `CatalogSource` you can pass the same argument that you use for catalog-queries. This makes it very flexible to limit relateable items by type, path, date etc.

For even more flexibility you can create your own dynamic vocabularies.

Accessing and displaying related items

One would think that it would be the easiest approach to simply use the render-method of the default-widget like we did in the chapter "Views II: A Default View for "Talk"". Sadly that is wrong. Adding the approriate code to te template:

would only render the UIDs of the related items:

This is not very useful but anyway it is very likely that you want to control closely how to render these items.

So we add a method to the view to return the related items so that we're able to render anyway we like.

```
def minions(self):
    """Returns a list of brains of related items."""
    results = []
    catalog = api.portal.get_tool('portal_catalog')
    for rel in self.context.underlings:
        if rel.isBroken():
            # skip broken relations
            continue
        # query by path so we don't have to wake up any objects
        brains = catalog(path={'query': rel.to_path, 'depth': 0})
        results.append(brains[0])
    return results
```

We use `rel.to_path()` and use the items path to query the catalog for its catalog-entry. This is much more efficient than using `rel.to_object()` since we don't have to wake up any objects. Setting `depth` to `0` will only return items with exactly this path, so it will always return a list with one item.

Note: Using the path sounds a little complicated and it would indeed be more convenient if a `RelationItem` would contain the `UID` (so we can query the catalog for that) or if the `portal_catalog` would index the `IntId`. But that's the way it is for now.

For reference look at how the default viewlet displays the information for related items stored by the behavior `IRelatedItems`. See how it does exactly the same in `related2brains`. This is the Python-path for the viewlet: `plone.app.layout.viewlets.content.ContentRelatedItems` This is the file-path for the template: `plone/app/layout/viewlets/document_relateditems.pt`

Creating Relationfields through the web

It is surprisingly easy to create RelationFields through the web

- In the dexterity schema-editor add a new field and select *Relation List* or *Relation Choice*, depending on wether you want to relate to multiple items or not.
- When configuring the field you can even select the content-type the relation should be limited to.

When you click on `Edit xml field model` you will see the fields in the xml-schema:

RelationChoice:

```
<field name="boss" type="z3c.relationfield.schema.RelationChoice">
  <description/>
  <required>False</required>
  <title>Boss</title>
</field>
```

RelationList:

```
<field name="underlings" type="z3c.relationfield.schema.RelationList">
  <description/>
  <required>False</required>
  <title>Underlings</title>
  <value_type type="z3c.relationfield.schema.RelationChoice">
    <title i18n:translate="">Relation Choice</title>
    <portal_type>
      <element>Document</element>
      <element>News Item</element>
    </portal_type>
  </value_type>
</field>
```

The stack

Relations are based on zc.relation. This package allows to store transitive and intransitive relationships. It allows for complex relationships and searches along them. Because of this functionality, the package is a bit complicated.

The package *zc.relation* provides its own catalog, a relation catalog. This is a storage optimized for the queries needed. *zc.relation* is sort of an outlier with regards to zope documentation. It has extensive documentation, with a good level of doctests for explaining things.

You can use *zc.relation* to store the objects and its relations directly into the catalog. But the additional packages that make up the relation functionality don't use the catalog this way.

We want to work with schemas to get auto generated forms. The logic for this is provided by the package z3c.relationfield. This package contains the RelationValue object and everything needed to define a relation schema, and all the code that is necessary to automatically update the catalog.

A RelationValue Object does not reference all objects directly. For the target, it uses an id it gets from the *IntId* Utility. This id allows direct recovery of the object. The source object stores it directly.

Widgets are provided by *plone.app.z3cform* and some converters are provided by *plone.app.relationfield*. The widget that Plone uses can also store objects directly. Because of this, the following happens when saving a relation via a form:

1. The html shows some nice representation of selectable objects.
2. When the user submits the form, selected items are submitted by their UUIDs.
3. The Widget retrieves the original object with the UUID.
4. Some datamanager gets another unique ID from an IntID Tool.
5. The same datamanager creates a RelationValue from this id, and stores this relation value on the source object.
6. Some Event handlers update the catalogs.

You could delete a Relation like this *delattr(rel.from_object, rel.from_attribute)*

This is a terrible idea by the way, because when you define in your schema that one can store multiple RelationValues, your Relation is stored in a list on this attribute.

Relations depend on a lot of infrastructure to work. This infrastructure in turn depends a lot on event handlers being thrown properly. When this is not the case things can break. Because of this, there is a method *isBroken* which you can use to check if the target is available.

There are alternatives to using Relations. You could instead just store the UUID of an object. But using real relations and the catalog allows for very powerful things. The simplest concrete advantage is the possibility to see what links to your object.

The builtin linkintegrity-feature of Plone 5 is also built using relations.

RelationValues

RelationValue objects have a fairly complete API. For both target and source, you can receive the IntId, the object and the path. On a RelationValue, the terms *source* and *target* aren't used. Instead, they are *from* and *to*. So the API for getting the target is:

- *to_id*
- *to_path*
- *to_object*

In addition, the relation value knows under which attribute it has been stored as *from_attribute*. It is usually the name of the field with which the relation is created. But it can also be the name of a relation that is created by code, e.g. linkintegrity-relations (*isReferencing*) or the relation between a working copy and the original (*iterate-working-copy*).

Accessing relations and backrelations from code

If you want to find out what objects are related to each other, you use the relation catalog. Here is a convenience-method that allows you to find all kinds of relations.

```
from zc.relation.interfaces import ICatalog
from zope.component import getUtility
from zope.intid.interfaces import IIntIds
from plone.app.linkintegrity.handlers import referencedRelationship


def example_get_backlinks(obj):
    backlinks = []
    for rel in get_backrelations(attribute=referencedRelationship):
        if rel.isBroken():
            backlinks.append(dict(href='',
                                  title='broken reference',
                                  relation=rel.from_attribute))
        else:
            obj = rel.from_object
            backlinks.append(dict(href=obj.absolute_url(),
                                  title=obj.title,
                                  relation=rel.from_attribute))
    return backlinks

def get_relations(obj, attribute=None, backrefs=False):
    """Get any kind of references and backreferences"""
    int_id = get_intid(obj)
    if not int_id:
        return retval

    relation_catalog = getUtility(ICatalog)
    if not relation_catalog:
        return retval

    query = {}
    if attribute:
        # Constrain the search for certain relation-types.
        query['from_attribute'] = attribute

    if backrefs:
        query['to_id'] = int_id
    else:
        query['from_id'] = int_id

    return relation_catalog.findRelations(query)
```

```
def get_backrelations(obj, attribute=None):
    return get_relations(obj, attribute=attribute, backrefs=True)


def get_intid(obj):
    """Return the intid of an object from the intid-catalog"""
    intids = component.queryUtility(IIntIds)
    if intids is None:
        return
    # check that the object has an intid, otherwise there's nothing to be done
    try:
        return intids.getId(obj)
    except KeyError:
        # The object has not been added to the ZODB yet
        return
```

Manage Settings with Registry, Controlpanels and Vocabularies

Get the code!

Get the code for this chapter (More info):

```
git checkout registry
```

In this part you will:

- Store a custom setting in a registry
- Create a controlpanel using z3c.form to allow setting that value

Topics covered:

- plone.app.registry
- controlpanels

The Registry

The registry is used to get and set values stored in records. Each record contains the actual value, as well as a field that describes the record in more detail. It has a nice dict-like API.

All global settings in Plone 5 are stored in the registry.

The registry itself is provided by plone.registry and the UI to interact with it by plone.app.registry

Almost all settings in `/plone_control_panel` are actually stored in the registry and can be modified using its UI directly.

Open http://localhost:8080/Plone/portal_registry and filter for `displayed_types`. You see can modify the content types that should be shown in the navigation and site map. The values are the same as in http://localhost:8080/Plone/@@navigation-controlpanel but the later form is customized for usability.

A setting

Let's store two values in the registry:

- The date of the conference
- Is talk submission open or closed

You cannot create values ttw, instead they need to be registered using Generic Setup.

Open the file `profiles/default/registry.xml`. You already registered several new settings in there:

- You enabled self registration
- You stored a site-logo
- You registered additional criteria useable for Collections

Adding the following code to `registry.xml`. This creates a new value in the registry upon installation of the package.

```xml
<record name="ploneconf.talk_submission_open">
  <field type="plone.registry.field.Bool">
    <title>Allow talk submission</title>
    <description>Allow the submission of talks for anonymous users</description>
    <required>False</required>
  </field>
  <value>False</value>
</record>
```

When creating a new site a lot of settings are created in the same way. See https://github.com/plone/Products.CMFPlone/blob/master/Products/CMFPlone/profiles/dependencies/registry.xml to see how `Products.CMFPlone` registers values.

```xml
<record name="ploneconf.date_of_conference">
  <field type="plone.registry.field.Date">
    <title>First day of the conference</title>
    <required>False</required>
  </field>
  <value>2016-10-17</value>
</record>
```

Accessing and modifying values in the registry

In python you can access the registry like this:

```python
from plone.registry.interfaces import IRegistry
from zope.component import getUtility

registry = getUtility(IRegistry)
start = registry.get('ploneconf.date_of_conference')
```

`plone.api` holds methods to make this even easier:

```python
from plone import api
api.portal.get_registry_record('ploneconf.date_of_conference')
api.portal.set_registry_record('ploneconf.talk_submission_open', True)
```

Add a custom controlpanel

When you want to add a custom controlpanel it is usually more convenient to register the fields not manually like above but as field in a schema, similar to a content-types schema.

For this you define a interface for the schema and a view that auto-generates a form from the schema. In `browser/configure.zcml` add:

```
<browser:page
    name="ploneconf-controlpanel"
    for="Products.CMFPlone.interfaces.IPloneSiteRoot"
    class=".controlpanel.PloneconfControlPanelView"
    permission="cmf.ManagePortal"
    />
```

Add a file `browser/controlpanel.py`:

```
# -*- coding: utf-8 -*-
from datetime import date
from plone.app.registry.browser.controlpanel import ControlPanelFormWrapper
from plone.app.registry.browser.controlpanel import RegistryEditForm
from plone.z3cform import layout
from zope import schema
from zope.interface import Interface


class IPloneconfControlPanel(Interface):

    date_of_conference = schema.Date(
        title=u'First day of the conference',
        required=False,
        default=date(2016, 10, 17),
    )

    talk_submission_open = schema.Bool(
        title=u'Allow talk submission',
        description=u'Allow the submission of talks for anonymous user',
        default=False,
        required=False,
    )


class PloneconfControlPanelForm(RegistryEditForm):
    schema = IPloneconfControlPanel
    schema_prefix = "ploneconf"
    label = u'Ploneconf Settings'


PloneconfControlPanelView = layout.wrap_form(
    PloneconfControlPanelForm, ControlPanelFormWrapper)
```

With this way of using fields you don't have to register the values in `registry.xml`, instead you have to register the interface:

```
<records interface="ploneconf.site.browser.controlpanel.IPloneconfControlPanel"
         prefix="ploneconf" />
```

After reinstalling the package (to load the registry-entry) you can access the controlpanel at http://localhost:8080/Plone/@@ploneconf-controlpanel.

To make it show up in the general controlpanel at http://localhost:8080/Plone/@@overview-controlpanel you have to register it with GenericSetup. Add a file `profiles/default/controlpanel.xml`:

```xml
<?xml version="1.0"?>
<object name="portal_controlpanel">
  <configlet
      title="Ploneconf Settings"
      action_id="ploneconf-controlpanel"
      appId="ploneconf-controlpanel"
      category="Products"
      condition_expr=""
      icon_expr=""
      url_expr="string:${portal_url}/@@ploneconf-controlpanel"
      visible="True">
    <permission>Manage portal</permission>
  </configlet>
</object>
```

Again, after applying the profile (reinstall the package or write a upgrade-step) your controlpanel shows up in http://localhost:8080/Plone/@@overview-controlpanel.

Vocabularies

Do you remember the field *rooms*? We provided several options to chose from. But who says that the next conference will have the same rooms? These values should be configurable by the admin. The admin could go to the dexterity-controlpanel and change the values but we will use a different approach. We will allow the rooms to be added in the controlpanel and use these values in the talk-schema by registering a vocabulary.

Add a new field to `IPloneconfControlPanel`:

```python
rooms = schema.Tuple(
    title=u'Available Rooms for the conference',
    default=(u'101', u'201', u'Auditorium'),
    missing_value=None,
    required=False,
    value_type=schema.TextLine()
)
```

Create a file `vocabularies.py` and write the vocabulary:

```python
# -*- coding: utf-8 -*-
from plone import api
from plone.i18n.normalizer.interfaces import IIDNormalizer
from zope.component import queryUtility
from zope.interface import implementer
from zope.schema.interfaces import IVocabularyFactory
from zope.schema.vocabulary import SimpleVocabulary

@implementer(IVocabularyFactory)
class RoomsVocabularyFactory(object):

    def __call__(self, context):
        values = api.portal.get_registry_record('ploneconf.rooms')
        normalizer = queryUtility(IIDNormalizer)
        items = [(normalizer.normalize(i), i) for i in values]
        return SimpleVocabulary.fromItems(items)

RoomsVocabulary = RoomsVocabularyFactory()
```

Note:

- *RoomsVocabulary* is a instance of `RoomsVocabularyFactory`.
- We normalize values to create a vocabulary since the value of a SimpleVocabulary has to be ASCII. We use one of many useful normalizers that Plone provides.

Register a vocabulary in `configure.zcml` as *ploneconf.site.vocabularies.Rooms*:

```
<utility
    name="ploneconf.site.vocabularies.Rooms"
    component="ploneconf.site.vocabularies.RoomsVocabulary" />
```

Use the vocabulary in the talk-schema. Edit `content/talk.xml`

```
<field name="room"
       type="zope.schema.Choice"
       form:widget="z3c.form.browser.radio.RadioFieldWidget"
       security:write-permission="cmf.ReviewPortalContent">
  <description></description>
  <title>Room</title>
  <vocabulary>ploneconf.site.vocabularies.Rooms</vocabulary>
</field>
```

Now a admin can configure the rooms available for the conference. We could use the same pattern for the fields *type_of_talk* and *audience*.

See also:

http://docs.plone.org/external/plone.app.dexterity/docs/advanced/vocabularies.html

Note: In a python-schema that would look like this:

```
directives.widget(room=RadioFieldWidget)
room = schema.Choice(
    title=_(u'Room'),
    vocabulary='ploneconf.site.vocabularies.Rooms',
    required=False,
)
```

Creating a Dynamic Front Page

Get the code!

Get the code for this chapter (More info):

```
git checkout frontpage
```

In this chapter we will:

- Create a standalone view used for the front page
- Show dynamic content
- Use ajax to load content
- Embed tweets about ploneconf

The topics we cover are:

- Standalone views
- Querying the catalog by date
- DRY
- macros
- patterns

The Front Page

Register the view in `browser/configure.zcml`:

```
<browser:page
    name="frontpageview"
    for="*"
    layer="ploneconf.site.interfaces.IPloneconfSiteLayer"
    class=".frontpage.FrontpageView"
    template="templates/frontpageview.pt"
    permission="zope2.View"
    />
```

Add the view to a file `browser/frontpage.py`. We want a list of all talks that happen today.

```
# -*- coding: utf-8 -*-
from plone import api
from Products.Five.browser import BrowserView

import datetime


class FrontpageView(BrowserView):
    """The view of the conference frontpage
    """

    def talks(self):
        """Get today's talks"""
        results = []
        today = datetime.date.today()
        brains = api.content.find(
            portal_type='talk',
            sort_on='start',
            sort_order='descending',
        )
        for brain in brains:
            if brain.start.date() == today:
                results.append({
                    'title': brain.Title,
                    'description': brain.Description,
                    'url': brain.getURL(),
                    'audience': ', '.join(brain.audience or []),
                    'type_of_talk': brain.type_of_talk,
                    'speaker': brain.speaker,
                    'room': brain.room,
                    'start': brain.start,
                    })
        return results
```

- We do not constrain the search to a certain folder to also find the party and the sprints.
- With `if brain.start.date() == today:` we test if the talk is today.
- It would be more effective to query the catalog for events that happen in the daterange between today and tomorrow:

```
today = datetime.date.today()
tomorrow = today + datetime.timedelta(days=1)
date_range_query = {'query': (today, tomorrow), 'range': 'min:max'}
brains = api.content.find(
    portal_type='talk',
    start=date_range_query,
    sort_on='start',
    sort_order='ascending'
)
```

- The `sort_on='start'` sorts the results returned by the catalog by start-date.
- By removing the `portal_type='talk'` from the query you could include other events in the schedule (like the party or sightseeing-tours). But you'd have to take care to not create AttributeErrors by accessing fields that are specific to talk. To work around that use `speaker = getattr(brain,'speaker',None)` and testing with `if speaker is not None:`
- The rest is identical to what the talklistview does.

The template

Create the template `browser/templates/frontpageview.pt` (for now without talks). Display the rich text field to allow the frontpage to be edited.

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en"
      metal:use-macro="context/main_template/macros/master"
      i18n:domain="ploneconf.site">
<body>

<metal:content-core fill-slot="content-core">

    <div id="parent-fieldname-text"
        tal:condition="python: getattr(context, 'text', None)"
        tal:content="structure python:context.text.output_relative_to(view.context)" /
↪>

</metal:content-core>

</body>
</html>
```

Now you could add the whole code that we used for the talklistview again. But instead we go D.R.Y. and reuse the talklistview by turning it into a macro.

Edit `browser/templates/talkslistview.pt` and wrap the list in a macro definition:

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en"
      metal:use-macro="context/main_template/macros/master"
      i18n:domain="ploneconf.site">
<body>
  <metal:content-core fill-slot="content-core">

  <metal:talklist define-macro="talklist">
  <table class="listing"
         id="talks"
         tal:define="talks python:view.talks()">
    <thead>
      <tr>
        <th>Title</th>
        <th>Speaker</th>
        <th>Audience</th>
        <th>Time</th>
        <th>Room</th>
      </tr>
    </thead>
    <tbody>
      <tr tal:repeat="talk talks">
        <td>
          <a href=""
             class="pat-contentloader"
             data-pat-contentloader="url:${python:talk['url']}?ajax_load=1;content:#content;target:.talkinfo > *"
             tal:attributes="href python:talk['url'];
                             title python:talk['description']"
             tal:content="python:talk['title']">
             The 7 sins of plone-development
          </a>
        </td>
        <td tal:content="python:talk['speaker']">
            Philip Bauer
        </td>
        <td tal:content="python:talk['audience']">
            Advanced
        </td>
        <td class="pat-moment"
            data-pat-moment="format:calendar"
            tal:content="python:talk['start']">
            Time
        </td>
        <td tal:content="python:talk['room']">
            101
        </td>
      </tr>
      <tr tal:condition="not:talks">
        <td colspan=5>
            No talks so far :-(
        </td>
      </tr>
    </tbody>
  </table>
  <div class="talkinfo"><span /></div>
  </metal:talklist>

  </metal:content-core>
</body>
</html>
```

Now use that macro in `browser/templates/frontpageview.pt`

```
<div class="col-lg-6">
    <h2>Todays Talks</h2>
    <div metal:use-macro="context/@@talklistview/talklist">
        Instead of this the content of the macro will appear...
    </div>
</div>
```

Calling that macro in python looks like this `metal:use-macro="python: context.restrictedTraverse('talklistview')['talklist']"`

Note: In `talklistview.pt` the call `view/talks"` calls the method `talks()` from the browser view `TalkListView` to get the talks. Reused as a macro on the frontpage it now uses the method `talks()` by the `frontpageView` to get a different list! It is not always smart to do that since you might want to display other data. E.g. for a list of todays talks you don't want show the date but only the time using `data-pat-moment="format:LT"` Also this frontpage will probably not win a beauty-contest. But that's not the task of this training.

Exercise 1

Change the link to open the talk-info in a modal.

Solution

```
<a href=""
   class="pat-plone-modal"
   tal:attributes="href string:${talk/url};
                   title talk/description"
   tal:content="talk/title">
   The 7 sins of plone development
</a>
```

Twitter

You might also want to embed a twitter feed into the page. Luckily twitter makes it easy to do that. When you browse to the twitter docs and learn how to create the appropriate snippet of code and paste it in the template wrapped in a `<div class="col-lg-6">...</div>` to have the talklist next to the feeds:

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en"
      metal:use-macro="context/main_template/macros/master"
      i18n:domain="ploneconf.site">
<body>

<metal:content-core fill-slot="content-core">

  <div id="parent-fieldname-text"
      tal:condition="python: getattr(context, 'text', None)"
      tal:content="structure python:context.text.output_relative_to(view.context)" />

  <div class="col-lg-6">
```

```
    <h2>Todays Talks</h2>
    <div metal:use-macro="context/@@talklistview/talklist">
        Instead of this the content of the macro will appear...
    </div>
  </div>

  <div class="col-lg-6">
    <a class="twitter-timeline"  href="https://twitter.com/search?q=ploneconf" data-
↪widget-id="786311347323535360">Tweets about ploneconf</a>
    <script>!function(d,s,id){var js,fjs=d.getElementsByTagName(s)[0],p=/^http:/.
↪test(d.location)?'http':'https';if(!d.getElementById(id)){js=d.createElement(s);js.
↪id=id;js.src=p+"://platform.twitter.com/widgets.js";fjs.parentNode.
↪insertBefore(js,fjs);}}(document,"script","twitter-wjs");</script>
  </div>

</metal:content-core>

</body>
</html>
```

Activating the view

The view is meant to be used with documents (or any other type that has a rich text field 'text'). The easiest way to use it is setting it as the default view for the Document that is currently the default page for the portal. By default that document has the id `front-page`.

You can either access it directly at http://localhost:8080/Plone/front-page or by disabling the default page for the portal and it should show up in the navigation. Try out the new view like this: http://localhost:8080/Plone/front-page/frontpageview.

To set that view by hand as the default view for `front-page` in the ZMI: http://localhost:8080/Plone/front-page/manage_propertiesForm. Add a new property `layout` and set it to `frontpageview`.

Done. This way you can still use the button *Edit* to edit the frontpage.

See also:

- Querying by date: http://docs.plone.org/develop/plone/searching_and_indexing/query.html#querying-by-date

Creating Reusable Packages

We already created the package `ploneconf.site` much earlier.

In this part you will:

- Build your own standalone egg.

Topics covered:

- `mr.bob`

Now you are going to create a feature that is completely independent of the ploneconf site and can be reused in other packages.

To make the distinction clear, this is not a package from the namespace `ploneconf` but from `starzel`.

We are going to add a voting behavior.

For this we need:

- A behavior that stores its data in annotations
- A viewlet to present the votes
- A bit of javascript
- A bit of css
- Some helper views so that the Javascript code can communicate with Plone

We move to the `src` directory and again use a script called `mrbob` from our project's `bin` directory and the template from `bobtemplates.plone` to create the package.

```
$ mkdir src
$ cd src
$ ../bin/mrbob -O starzel.votable_behavior bobtemplates:plone_addon
```

We press `Enter` to all questions *except* our personal data and the Plone version. Here we enter `5.0a3`.

More Complex Behaviors

In this part you will:

- Write an annotation

Topics covered:

- Annotation Marker Interfaces

Using Annotations

We are going to store the information in an annotation. Not because it is needed but because you will find code that uses annotations and need to understand the implications.

Annotations in Zope/Plone mean that data won't be stored directly on an object but in an indirect way and with namespaces so that multiple packages can store information under the same attribute, without colliding.

So using annotations avoids namespace conflicts. The cost is an indirection. The dictionary is persistent so it has to be stored separately. Also, one could give attributes a name containing a namespace prefix to avoid naming collisions.

Using Schema

The attribute where we store our data will be declared as a schema field. We mark the field as an omitted field (using schema directive similar to `read_permission` or `widget`), because we are not going to create `z3c.form` widgets for entering or displaying them. We do provide a schema, because many other packages use the schema information to get knowledge of the relevant fields.

For example, when files were migrated to blobs, new objects had to be created and every schema field was copied. The code can't know about our field, except if we provide schema information.

Writing Code

To start, we create a directory `behavior` with an empty `behavior/__init__.py` file.

Next we must, as always, register our ZCML.

First, add the information that there will be another ZCML file in `configure.zcml`

```
<configure xmlns="...">

  ...
  <include package=".behavior" />
  ...

</configure>
```

Next, create behavior/configure.zcml

```
<configure
    xmlns="http://namespaces.zope.org/zope"
    xmlns:plone="http://namespaces.plone.org/plone">

  <plone:behavior
      title="Voting"
      description="Allow voting for an item"
      provides="starzel.votable_behavior.interfaces.IVoting"
      factory=".voting.Vote"
      marker="starzel.votable_behavior.interfaces.IVotable"
      />

</configure>
```

There are some important differences to our first behavior:

- There is a marker interface
- There is a factory

The factory is a class that provides the behavior logic and gives access to the attributes we provide. Factories in Plone/Zope land are retrieved by adapting an object to an interface. If you want your behavior, you would write `IVoting(object)`

But in order for this to work, your object may *not* be implementing the IVoting interface, because if it did, `IVoting(object)` would return the object itself! If I need a marker interface for objects providing my behavior, I must provide one, for this we use the marker attribute. My object implements `IVotable` and because of this, we can write views and viewlets just for this content type.

The interfaces need to be written, in our case into a file `interfaces.py`:

```
# encoding=utf-8
from plone import api
from plone.autoform import directives
from plone.autoform.interfaces import IFormFieldProvider
from plone.supermodel import model
from plone.supermodel.directives import fieldset
from zope import schema
from zope.interface import alsoProvides
from zope.interface import Interface

class IVotableLayer(Interface):
    """Marker interface for the Browserlayer
    """

# Ivotable is the marker interface for contenttypes who support this behavior
class IVotable(Interface):
    pass

```

```
# This is the behaviors interface. When doing IVoting(object), you receive an
# adapter
class IVoting(model.Schema):
    if not api.env.debug_mode():
        directives.omitted("votes")
        directives.omitted("voted")

    fieldset(
        'debug',
        label=u'debug',
        fields=('votes', 'voted'),
    )

    votes = schema.Dict(title=u"Vote info",
                        key_type=schema.TextLine(title=u"Voted number"),
                        value_type=schema.Int(title=u"Voted so often"),
                        required=False)
    voted = schema.List(title=u"Vote hashes",
                        value_type=schema.TextLine(),
                        required=False)

    def vote(request):
        """
        Store the vote information, store the request hash to ensure
        that the user does not vote twice
        """

    def average_vote():
        """
        Return the average voting for an item
        """

    def has_votes():
        """
        Return whether anybody ever voted for this item
        """

    def already_voted(request):
        """
        Return the information wether a person already voted.
        This is not very high level and can be tricked out easily
        """

    def clear():
        """
        Clear the votes. Should only be called by admins
        """

alsoProvides(IVoting, IFormFieldProvider)
```

This is a lot of code. The IVotableLayer we will need later for viewlets and browser views. Let's add it right here. The IVotable interface is the simple marker interface. It will only be used to bind browser views and viewlets to contenttypes that provide our behavior, so no code needed.

The IVoting class is more complex, as you can see. While IVoting is just an interface, we use `plone.supermodel.model.Schema` for advanced dexterity features. Zope.schema provides no means for hiding fields. The directives `form.omitted` from `plone.autoform` allow us to annotate this additional information so that the autoform renderers for forms can use the additional information.

We make this omit conditional. If we run Plone in debug mode, we will be able to see the internal data in the edit form.

We create minimal schema fields for our internal data structures. For a small test, I removed the form omitted directives and opened the edit view of a talk that uses the behavior. After seeing the ugliness, I decided that I should provide at least minimum of information. Titles and required are purely optional, but very helpful if the fields won't be omitted, something that can be helpful when debugging the behavior. Later, when we implement the behavior, the `votes` and `voted` attributes are implemented in such a way that you can't just modify these fields, they are read only.

Then we define the API that we are going to use in browser views and viewlets.

The last line ensures that the schema fields are known to other packages. Whenever some code wants all schemas from an object, it receives the schema defined directly on the object and the additional schemata. Additional schemata are compiled by looking for behaviors and whether they provide the `IFormFieldProvider` functionality. Only then the fields are known as schema fields.

Now the only thing that is missing is the behavior, which we must put into `behavior/voting.py`

```python
# encoding=utf-8
from hashlib import md5
from persistent.dict import PersistentDict
from persistent.list import PersistentList
from zope.annotation.interfaces import IAnnotations

KEY = "starzel.votable_behavior.behavior.voting.Vote"


class Vote(object):
    def __init__(self, context):
        self.context = context
        annotations = IAnnotations(context)
        if KEY not in annotations.keys():
            annotations[KEY] = PersistentDict({
                "voted": PersistentList(),
                'votes': PersistentDict()
                })
        self.annotations = annotations[KEY]

    @property
    def votes(self):
        return self.annotations['votes']

    @property
    def voted(self):
        return self.annotations['voted']
```

In our `__init__` method we get *annotations* from the object. We look for data with a specific key.

The key in this example is the same as what I would get with `__name__+Vote.__name__`. But we won't create a dynamic name, this would be very clever and clever is bad.

By declaring a static name, we won't run into problems if we restructure the code.

You can see that we initialize the data if it doesn't exist. We work with PersistentDict and PersistentList. To understand why we do this, it is important to understand how the ZODB works.

See also:

The ZODB can store objects. It has a special root object that you will never touch. Whatever you store there, will be part of the root object, except if it is an object subclassing `persistent.Persistent` Then it will be stored independently.

Zope/ZODB Persistent objects note when you change an attribute on it and mark itself as changed. Changed objects will be saved to the database. This happens automatically. Each request begins a transaction and after our code runs and the Zope Server is preparing to send back the response we generated, the transaction will be committed and everything we changed will be saved.

Now, if have a normal dictionary on a persistent object, and you will only change the dictionary, the persistent object has no way to know if the dictionary has been changed. This happens from time to time.

So one solution is to change the special attribute _p_changed to True on the persistent object, or to use a PersistentDict. That is what we are doing here.

An important thing to note about PersistentDict and PersistentList is that they cannot handle write conflicts. What happens if two users rate the same content independently at the same time? In this case, a database conflict will occur because there is no way for Plone to know how to handle the concurrent write access. Although this is rather unlikely during this training, it is a very common problem on high traffic websites.

You can find more information in the documentation of the ZODB, in particular Rules for Persistent Classes

Next we provide the internal fields via properties. Using this form of property makes them read only properties, as we did not define write handlers. We don't need them so we won't add them.

As you have seen in the Schema declaration, if you run your site in debug mode, you will see an edit field for these fields. But trying to change these fields will throw an exception.

Let's continue with this file:

```
    def _hash(self, request):
        """
        This hash can be tricked out by changing IP addresses and might allow
        only a single person of a big company to vote
        """
        hash_ = md5()
        hash_.update(request.getClientAddr())
        for key in ["User-Agent", "Accept-Language",
                    "Accept-Encoding"]:
            hash_.update(request.getHeader(key))
        return hash_.hexdigest()

    def vote(self, vote, request):
        if self.already_voted(request):
            raise KeyError("You may not vote twice")
        vote = int(vote)
        self.annotations['voted'].append(self._hash(request))
        votes = self.annotations['votes']
        if vote not in votes:
            votes[vote] = 1
        else:
            votes[vote] += 1

    def average_vote(self):
        if not has_votes(self):
            return 0
        total_votes = sum(self.annotations['votes'].values())
        total_points = sum([vote * count for (vote, count) in
                            self.annotations['votes'].items()])
        return float(total_points) / total_votes

    def has_votes(self):
        return len(self.annotations.get('votes', [])) != 0

    def already_voted(self, request):
        return self._hash(request) in self.annotations['voted']

    def clear(self):
        annotations = IAnnotations(self.context)
        annotations[KEY] = PersistentDict({'voted': PersistentList(),
                                           'votes': PersistentDict()})
        self.annotations = annotations[KEY]
```

We start with a little helper method which is not exposed via the interface. We don't want people to vote twice. There are many ways to ensure this and each one has flaws.

We chose this way to show you how to access information from the request the browser of the user sent to us. First, we get the ip of the user, then we access a small set of headers from the user's browser and generate an md5 checksum of this.

The vote method wants a vote and a request. We check the preconditions, then we convert the vote to an integer, store the request to `voted` and the votes into the `votes` dictionary. We just count there how often any vote has been given.

Everything else is just python.

Exercises

Exercise 1

Refactor the voting behavior so that it uses *BTrees* instead of *PersistentDict* and *PersistentList*. Use *OOBTree* to replace *PersistentDict* and *OIBTree* to replace *PersistentList*.

Solution

change `behavior/voting.py`

```
# encoding=utf-8
from hashlib import md5
from BTrees.OOBTree import OOBTree
from BTrees.OIBTree import OIBTree
from zope.annotation.interfaces import IAnnotations

KEY = "starzel.votable_behavior.behavior.voting.Vote"


class Vote(object):
    def __init__(self, context):
        self.context = context
        annotations = IAnnotations(context)
        if KEY not in annotations.keys():
            annotations[KEY] = OOBTree()
            annotations[KEY]['voted'] = OIBTree()
            annotations[KEY]['votes'] = OOBTree()
        self.annotations = annotations[KEY]

    ...

    def vote(self, vote, request):
        if self.already_voted(request):
            raise KeyError("You may not vote twice")
        vote = int(vote)
        self.annotations['voted'].insert(
            self._hash(request),
            len(self.annotations['voted']))
        votes = self.annotations['votes']
        if vote not in votes:
            votes[vote] = 1
        else:
            votes[vote] += 1

    ...

    def clear(self):
        annotations = IAnnotations(self.context)
        annotations[KEY] = OOBTree()
        annotations[KEY]['voted'] = OIBTree()
        annotations[KEY]['votes'] = OOBTree()
        self.annotations = annotations[KEY]
```

Exercise 2

Write a unit test that simulates concurrent voting. The test should raise a *ConflictError* on the original voting behavior implementation. The solution from the first exercise should pass. Look at the file *ZODB/ConflictResolution.txt* in the *ZODB3* egg for how to create a suitable test fixture for conflict testing. Look at the test code in *zope.annotation* for how to create annotatable dummy content. You will also have to write a 'request' dummy that mocks the *getClientAddr* and *getHeader* methods of Zope's HTTP request object to make the *_hash* method of the voting behavior work.

Solution

There are no tests for *starzel.votablebehavior* at all at the moment. But you can refer to chapter 22 for how to setup unit testing for a package. Put the particular test for this exercise into a file named `starzel.votable_behavior/starzel/votable_behavior/tests/test_voting`. Remember you need an empty `__init__.py` file in the `tests` directory to make testing work. You also need to add *starzel.votable_behavior* to *test-eggs* in `buildout.cfg` and re-run buildout.

```
import unittest
import tempfile
import ZODB
import transaction
from persistent import Persistent
from zope.interface import implements
from zope.annotation.interfaces import IAttributeAnnotatable
from zope.annotation.attribute import AttributeAnnotations


class Dummy(Persistent):
    implements(IAttributeAnnotatable)


class RequestDummy(object):

    def __init__(self, ip, headers=None):
        self.ip = ip
        if headers is not None:
            self.headers = headers
        else:
            self.headers = {
                'User-Agent': 'foo',
                'Accept-Language': 'bar',
                'Accept-Encoding': 'baz'
                }

    def getClientAddr(self):
        return self.ip

    def getHeader(self, key):
        return self.headers[key]


class VotingTests(unittest.TestCase):

    def test_voting_conflict(self):
        from starzel.votable_behavior.behavior.voting import Vote
        dbname = tempfile.mktemp()
        db = ZODB.DB(dbname)
        tm_A = transaction.TransactionManager()
        conn_A = db.open(transaction_manager=tm_A)
        p_A = conn_A.root()['voting'] = Vote(AttributeAnnotations(Dummy()))
        tm_A.commit()
        # Now get another copy of 'p' so we can make a conflict.
        # Think of `conn_A` (connection A) as one thread, and
        # `conn_B` (connection B) as a concurrent thread.  `p_A`
        # is a view on the object in the first connection, and `p_B`
        # is a view on *the same persistent object* in the second connection.
        tm_B = transaction.TransactionManager()
        conn_B = db.open(transaction_manager=tm_B)
        p_B = conn_B.root()['voting']
        assert p_A.context.obj._p_oid == p_B.context.obj._p_oid
        # Now we can make a conflict, and see it resolved (or not)
        request_A = RequestDummy('192.168.0.1')
        p_A.vote(1, request_A)
        request_B = RequestDummy('192.168.0.5')
        p_B.vote(2, request_B)
        tm_B.commit()
        tm_A.commit()
```

A Viewlet for the Votable Behavior

Voting Viewlet

In this part you will:

- write the viewlet template
- add jquery include statements
- saving the vote on the object using annotations

Topics covered:

- Viewlets

- Javascript inclusion

Earlier we added the logic that saves votes on the objects. We now create the user interface for it.

Since we want to use the UI on more than one page (not only the talk view but also the talk listing) we need to put it somewhere.

- To handle the user input we don't use a form but links and ajax.
- The voting itself is a fact handled by another view

We register the viewlet in browser/configure.zcml.

```
<configure xmlns="http://namespaces.zope.org/zope"
    xmlns:browser="http://namespaces.zope.org/browser">

    ...

  <browser:viewlet
    name="voting"
    for="starzel.votable_behavior.interfaces.IVoting"
    manager="plone.app.layout.viewlets.interfaces.IBelowContentTitle"
    layer="..interfaces.IVotableLayer"
    class=".viewlets.Vote"
    template="templates/voting_viewlet.pt"
    permission="zope2.View"
    />

    ....

</configure>
```

We extend the file browser/viewlets.py

```
from plone.app.layout.viewlets import common as base


class Vote(base.ViewletBase):
    pass
```

This will add a viewlet to a slot below the title and expect a template voting_viewlet.pt in a folder browser/templates.

Let's create the file browser/templates/voting_viewlet.pt without any logic

```
<div class="voting">
    Wanna vote? Write code!
</div>

<script type="text/javascript">
  jq(document).ready(function(){
    // please add some jQuery-magic
  });
</script>
```

- restart Plone
- show the viewlet

Writing the Viewlet code

Update the viewlet to contain the necessary logic in `browser/viewlets`

```python
from plone.app.layout.viewlets import common as base
from Products.CMFCore.permissions import ViewManagementScreens
from Products.CMFCore.utils import getToolByName

from starzel.votable_behavior.interfaces import IVoting


class Vote(base.ViewletBase):

    vote = None
    is_manager = None

    def update(self):
        super(Vote, self).update()

        if self.vote is None:
            self.vote = IVoting(self.context)
        if self.is_manager is None:
            membership_tool = getToolByName(self.context, 'portal_membership')
            self.is_manager = membership_tool.checkPermission(
                ViewManagementScreens, self.context)

    def voted(self):
        return self.vote.already_voted(self.request)

    def average(self):
        return self.vote.average_vote()

    def has_votes(self):
        return self.vote.has_votes()
```

The template

And extend the template in `browser/templates/voting_viewlet.pt`

```html
<tal:snippet omit-tag="">
  <div class="voting">
    <div id="current_rating" tal:condition="viewlet/has_votes">
      The average vote for this talk is <span tal:content="viewlet/average">200</span>
    </div>
    <div id="alreadyvoted" class="voting_option">
      You already voted this talk. Thank you!
    </div>
    <div id="notyetvoted" class="voting_option">
      What do you think of this talk?
      <div class="votes"><span id="voting_plus">+1</span> <span id="voting_neutral">0</span> <span id="voting_negative">-1</span>
      </div>
    </div>
    <div id="no_ratings" tal:condition="not: viewlet/has_votes">
      This talk has not been voted yet. Be the first!
    </div>
    <div id="delete_votings" tal:condition="viewlet/is_manager">
```

```
      Delete all votes
    </div>
    <div id="delete_votings2" class="areyousure warning"
         tal:condition="viewlet/is_manager"
         >
      Are you sure?
    </div>
    <a href="#" class="hiddenStructure" id="context_url"
       tal:attributes="href context/absolute_url"></a>
    <span id="voted" tal:condition="viewlet/voted"></span>
  </div>
  <script type="text/javascript">
    $(document).ready(function(){
      starzel_votablebehavior.init_voting_viewlet($(".voting"));
    });
  </script>
</tal:snippet>
```

We have many small parts, most of which will be hidden by javascript unless needed. By providing all this status information in HTML, we can use standard translation tools to translate. Translating strings in javascript requires extra work.

We need some css that we store in `static/starzel_votablebehavior.css`

```
.voting {
    float: right;
    border: 1px solid #ddd;
    background-color: #DDDDDD;
    padding: 0.5em 1em;
}

.voting .voting_option {
    display: none;
}

.areyousure {
    display: none;
}

.voting div.votes span {
    border: 0 solid #DDDDDD;
    cursor: pointer;
    float: left;
    margin: 0 0.2em;
    padding: 0 0.5em;
}

.votes {
    display: inline;
    float: right;
}

.voting #voting_plus {
    background-color: LimeGreen;
}

.voting #voting_neutral {
    background-color: yellow;
}

.voting #voting_negative {
    background-color: red;
}
```

The javascript code

To make it work in the browser, some javascript `static/starzel_votablebehavior.js`

```
/*global location: false, window: false, jQuery: false */
(function ($, starzel_votablebehavior) {
    "use strict";
    starzel_votablebehavior.init_voting_viewlet = function (context) {
        var notyetvoted = context.find("#notyetvoted"),
            alreadyvoted = context.find("#alreadyvoted"),
            delete_votings = context.find("#delete_votings"),
            delete_votings2 = context.find("#delete_votings2");

        if (context.find("#voted").length !== 0) {
            alreadyvoted.show();
        } else {
            notyetvoted.show();
        }

        function vote(rating) {
            return function inner_vote() {
                $.post(context.find("#context_url").attr('href') + '/vote', {
                    rating: rating
                }, function () {
                    location.reload();
                });
            };
        }

        context.find("#voting_plus").click(vote(1));
        context.find("#voting_neutral").click(vote(0));
        context.find("#voting_negative").click(vote(-1));

        delete_votings.click(function () {
            delete_votings2.toggle();
        });
        delete_votings2.click(function () {
            $.post(context.find("#context_url").attr("href") + "/clearvotes",␣
↪function () {
                location.reload();
            });
        });
    };
}(jQuery, window.starzel_votablebehavior = window.starzel_votablebehavior || {}));
```

This js code adheres to crockfort jshint rules, so all variables are declared at the beginning of the method. We show and hide quite a few small html elements here.

Writing 2 simple view helpers

Our javascript code communicates with our site by calling views that don't exist yet. These Views do not need to render html, but should return a valid status. Exceptions set the right status and aren't being shown by javascript, so this will suit us fine.

As you might remember, the `vote` method might return an exception, if somebody votes twice. We do not catch this exception. The user will never see this exception.

See also:

Catching exceptions contain a gotcha for new developers.

```
try:
    something()
except:
    fix_something()
```

Zope claims some exceptions for itself. It needs them to work correctly.

For example, if two requests try to modify something at the same time, one request will throw an exception, a `ConflictError`.

Zope catches the exception, waits for a random amount of time, and tries to process the request again, up to three times. If you catch that exception, you are in trouble, so don't do that. Ever.

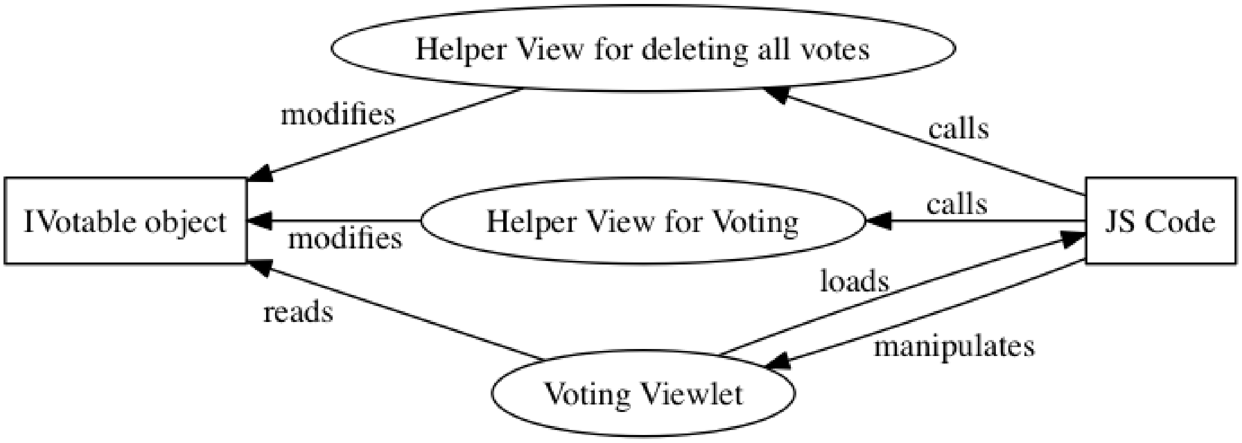
As so often, we must extend `browser/configure.zcml`:

```
...

<browser:page
  name="vote"
  for="starzel.votable_behavior.interfaces.IVotable"
  layer="..interfaces.IVotableLayer"
  class=".vote.Vote"
  permission="zope2.View"
  />

<browser:page
  name="clearvotes"
  for="starzel.votable_behavior.interfaces.IVotable"
  layer="..interfaces.IVotableLayer"
  class=".vote.ClearVotes"
  permission="zope2.ViewManagementScreens"
  />

...
```

Then we add our simple views into the file `browser/vote.py`

```
from zope.publisher.browser import BrowserPage

from starzel.votable_behavior.interfaces import IVoting


class Vote(BrowserPage):

    def __call__(self, rating):
        voting = IVoting(self.context)
        voting.vote(rating, self.request)
        return "success"


class ClearVotes(BrowserPage):

    def __call__(self):
        voting = IVoting(self.context)
        voting.clear()
        return "success"
```

A lot of moving parts have been created. Here is a small overview:



Making Our Package Reusable

In this part you will:

- Add Permissions

Topics covered:

- Permissions

The package contains some problems.

- No permission settings, Users can't customize who and when users can vote
- We do things, but don't trigger events. Events allow others to react.

Adding permissions

Permissions have a long history, there are two types of permissions.

In Zope2, a permission was just a string.

In ZTK, a permission is an object that gets registered as a Utility.

We must support both, in some cases we have to reference the permission by their Zope2 version, in some by their ZTK Version.

Luckily, there is a zcml statement to register a permission both ways in one step.

See also:

The configuration registry was meant to solve a problem, but we will now stumble over a problem that did not get resolved properly.

Our permission is a utility. Our browser views declare this permission as a requirement for viewing them.

When our browser views get registered, the permissions must exist already. If you try to register the permissions after the views, Zope won't start because it doesn't know about the permissions.

Let's modify the file `configure.zcml`

```
<configure xmlns="...">

  <includeDependencies package="." />

  <permission
      id="starzel.votable_behavior.view_vote"
      title="starzel.votable_behavior: View Vote"
      />

  <permission
      id="starzel.votable_behavior.do_vote"
      title="starzel.votable_behavior: Do Vote"
      />

  <include package=".browser" />

  ...

</configure>
```

In some places we have to reference the Zope 2 permission strings. It is best practice to provide a static variable for this.

We provide this in `__init__.py`

```
...
DoVote = 'starzel.votable_behavior: Do Vote'
ViewVote = 'starzel.votable_behavior: View Vote'
```

Using our permissions

As you can see, we created two permissions, one for voting, one for viewing the votes.

If a user is not allowed to see the votes, she does not need access to the vote viewlet.

While we are at it, if a user can't vote, she needs no access to the helper view to actually submit a vote.

We can add this restriction to `browser/configure.zcml`

```
<configure
  xmlns="http://namespaces.zope.org/zope"
  xmlns:browser="http://namespaces.zope.org/browser"
  i18n_domain="starzel.votable_behavior">

  <browser:viewlet
    name="voting"
    for="starzel.votable_behavior.interfaces.IVotable"
    manager="plone.app.layout.viewlets.interfaces.IBelowContentTitle"
    template="templates/voting_viewlet.pt"
    layer="..interfaces.IVotableLayer"
    class=".viewlets.Vote"
    permission="starzel.votable_behavior.view_vote"
    />

  <browser:page
    name="vote"
    for="starzel.votable_behavior.interfaces.IVotable"
    layer="..interfaces.IVotableLayer"
    class=".vote.Vote"
    permission="starzel.votable_behavior.do_vote"
    />

  ...

</configure>
```

We are configuring components, so we use the component name of the permission, which is the `id` part of the declaration we added earlier.

See also:

So, what happens if we do not protect the browser view to vote?

The person could still vote, by handcrafting the URL. Browser Views run code without any restriction, it is your job to take care of security.

But... if a person has no access to the object at all, maybe because the site is configured that Anonymous users cannot access private objects, the unauthorized users will not be able to submit a vote.

That is because Zope checks security permissions when trying to find the right object. If it can't find the object due to security constraints not met, no view ill ever be called, because that would have been the next step.

We now protect our views and viewlets. We still show the option to vote though.

We must add a condition in our page template, and we must provide the condition information in our viewlet class.

Lets move on to `browser/viewlets.py`

```
...

from starzel.votable_behavior import DoVote


class Vote(base.ViewletBase):

     ...
     can_vote = None

    def update(self):

        ...

        if self.is_manager is None:
            membership_tool = getToolByName(self.context, 'portal_membership')
            self.is_manager = membership_tool.checkPermission(
                ViewManagementScreens, self.context)
            self.can_vote = membership_tool.checkPermission(
                DoVote, self.context)
```

```

...
```

And the template in `browser/templates/voting_viewlet.pt`

```
<tal:snippet omit-tag="">
  <div class="voting">

    ...

    <div id="notyetvoted" class="voting_option"
            tal:condition="view/can_vote">
      What do you think of this talk?
      <div class="votes"><span id="voting_plus">+1</span> <span id="voting_neutral">0</span> <span id="voting_negative">-1</span>
      </div>
    </div>
    <div id="no_ratings" tal:condition="not: view/has_votes">
      This talk has not been voted yet.<span tal:omit-tag="" tal:condition="view/can_vote"> Be the first!</span>
    </div>

  ...

  </div>

...

</tal:snippet>
```

Sometimes subtle bugs come up because of changes. In this case I noticed that I should only prompt people to vote if they are allowed to vote!

Provide defaults

Are we done yet? Who may vote now?

We have to tell that someone.

Who has which permissions is managed in Zope. This is persistent, and persistent configuration is handled by GenericSetup.

The persistent configuration is managed in another file: `profiles/default/rolemap.xml`

```
<?xml version="1.0"?>
<rolemap>
  <permissions>
    <permission name="starzel.votable_behavior: View Vote" acquire="True">
      <role name="Anonymous"/>
    </permission>
    <permission name="starzel.votable_behavior: Do Vote" acquire="True">
      <role name="Anonymous"/>
    </permission>
  </permissions>
</rolemap>
```

Using starzel.votable_behavior in ploneconf.site

In this part you will:

- Integrate your own third party package into your site.

Topics covered:

- Permissions
- Workflows

Get the code!

Get the code for this chapter (More info) using this command in the buildout directory:

```
TODO
```

- We want to use the votable behavior, so that our reviewers can vote.
- To show how to use events, we are going to auto-publish talks that have reached a certain rating.
- We are not going to let everybody vote everything.

First, we must add our package as a dependency to ploneconf.site.

We do this in two locations. The egg description `setup.py` needs `starzel.votable_behavior` as a dependency. Else no source code will be available.

The persistent configuration needs to be installed when we install our site. This is configured in GenericSetup.

We start by editing `setup.py`

```
...
zip_safe=False,
install_requires=[
    'setuptools',
    'plone.app.dexterity [relations]',
    'plone.app.relationfield',
    'plone.namedfile [blobs]',
    'starzel.votable_behavior',
    # -*- Extra requirements: -*-
],
...
```

Next up we modify `profiles/default/metadata.xml`

```
<metadata>
  <version>1002</version>
    <dependencies>
      <dependency>profile-starzel.votable_behavior:default</dependency>
    </dependencies>
</metadata>
```

... only:: not presentation

> What a weird name. profile- is a prefix you will always need nowadays. Then comes the egg name, and the part after the colon is the name of the profile. The name of the profile is defined in zcml. So far I've stumbled over only one package where the profile directory name was different than the GenericSetup Profile name.

> Now the package is there, but nothing is votable. That is because no content type declares to use this behavior. We can add this behavior via the control panel, export the settings and store it in our egg. Let's just add it by hand now.

To add the behavior to talks, we do this in `profiles/default/types/talk.xml`

Note: After changing the `metadata.xml` you have to restart your site since unlike other GenericSetup XML files that file is cached.

Managing dependencies in `metadata.xml` is good practice. We can't rely on remembering what we'd have to do by hand. In Plone 4 we should also have added `<dependency>profile-plone.app.contenttypes:plone-content</dependency>` like the documentation for plone.app.contenttypes recommends.

Read more: http://docs.plone.org/develop/addons/components/genericsetup.html#dependencies

```xml
<property name="behaviors">
  <element value="plone.app.dexterity.behaviors.metadata.IDublinCore"/>
  <element value="plone.app.content.interfaces.INameFromTitle"/>
  <element value="starzel.votable_behavior.interfaces.IVoting"/>
</property>
```

... only:: not presentation

> Now you can reinstall your Plone site.
>
> Everybody can now vote on talks. That's not what we wanted. We only want reviewers to vote on *pending* Talks. This means the permission has to change depending on the workflow state. Luckily, workflows can be configured to do just that. Since Talks already have their own workflow we also won't interfere with other content.
>
> First, we have to tell the workflow that it will be managing more permissions. Next, for each state we have to configure which role has the two new permissions.
>
> That is a very verbose configuration, maybe you want to do it in the web interface and export the settings. Whichever way you choose, it is easy to make a simple mistake. I will just present the XML way here.

The config for the Workflow is in `profiles/default/workfows/talks_workflow.xml`

```xml
<?xml version="1.0"?>
<dc-workflow workflow_id="talks_workflow" title="Talks Workflow" description=" -␣
↪Simple workflow that is useful for basic web sites. - Things start out as private,␣
↪and can either be submitted for review, or published directly. - The creator of a␣
↪content item can edit the item even after it is published." state_variable="review_
↪state" initial_state="private" manager_bypass="False">
 <permission>Access contents information</permission>
 <permission>Change portal events</permission>
 <permission>Modify portal content</permission>
 <permission>View</permission>
 <permission>starzel.votable_behavior: View Vote</permission>
 <permission>starzel.votable_behavior: Do Vote</permission>
 <state state_id="pending" title="Pending review">
  <description>Waiting to be reviewed, not editable by the owner.</description>
  ...
  <permission-map name="starzel.votable_behavior: View Vote" acquired="False">
   <permission-role>Site Administrator</permission-role>
   <permission-role>Manager</permission-role>
   <permission-role>Reviewer</permission-role>
  </permission-map>
  <permission-map name="starzel.votable_behavior: Do Vote" acquired="False">
   <permission-role>Site Administrator</permission-role>
   <permission-role>Manager</permission-role>
   <permission-role>Reviewer</permission-role>
  </permission-map>
  ...
 </state>
 <state state_id="private" title="Private">
  <description>Can only be seen and edited by the owner.</description>
  ...
  <permission-map name="starzel.votable_behavior: View Vote" acquired="False">
   <permission-role>Site Administrator</permission-role>
   <permission-role>Manager</permission-role>
  </permission-map>
  <permission-map name="starzel.votable_behavior: Do Vote" acquired="False">
   <permission-role>Site Administrator</permission-role>
   <permission-role>Manager</permission-role>
  </permission-map>
  ...
 </state>
 <state state_id="published" title="Published">
  <description>Visible to everyone, editable by the owner.</description>
  ...
  <permission-map name="starzel.votable_behavior: View Vote" acquired="False">
   <permission-role>Site Administrator</permission-role>
   <permission-role>Manager</permission-role>
  </permission-map>
  <permission-map name="starzel.votable_behavior: Do Vote" acquired="False">
  </permission-map>
  ...
 </state>
  ...
</dc-workflow>
```

We have to reinstall our product again.

But this time, this is not enough. Permissions get updated on workflow changes. As long as a workflow change didn't happen, the talks have the same permissions as ever.

Luckily, there is a button for that in the ZMI Workflow view *Update security settings*.

After clicking on this, only managers and Reviewers can see the Voting functionality.

Lastly, we add our silly function to auto-approve talks.

You quickly end up writing many event handlers, so we put everything into a directory for eventhandlers.

For the events we need an `events` directory.

Create the `events` directory and add an empty `events/__init__.py` file.

Next, register the events directory in `configure.zcml`

```
<include package=".events" />
```

Now write the ZCML configuration for the events into `events/configure.zcml`

```
<configure
    xmlns="http://namespaces.zope.org/zope">

  <subscriber
```

```
    for="starzel.votable_behavior.interfaces.IVotable
         zope.lifecycleevent.IObjectModifiedEvent"
    handler=".votable.votable_update"
    />

</configure>
```

This looks like a MultiAdapter. We want to get notified when an IVotable object gets modified. Our method will receive the votable object and the event itself.

And finally, our event handler in `events/votable.py`

```
from plone.api.content import transition
from plone.api.content import get_state
from starzel.votable_behavior.interfaces import IVoting


def votable_update(votable_object, event):
    votable = IVoting(votable_object)
    if get_state(votable_object) == 'pending':
        if votable.average_vote() > 0.5:
            transition(votable_object, transition='publish')
```

We are using a lot of plone api here. Plone API makes the code a breeze. Also, there is nothing really interesting. We will only do something if the workflow state is pending and the average vote is above 0.5. As you can see, the `transition` Method does not want the target state, but the transition to move the state to the target state.

There is nothing special going on.

Releasing Your Code

- zest.releaser
- pypi-test egg deployment

We finally have some working code! Depending on your policies, you need repeatable deployments and definitive versions of software. That means you don't just run your production site with your latest source code from your source repository. You want to work with eggs.

Making eggs is easy, making them properly not so much. There are a number of good practices that you should ensure. Let's see. You want to have a sensible version number. By looking at the version number alone one should get a good idea how many changes there are (semantic version number scheme). Of course you always document everything, but for upgrades it is even more important to have complete changelogs.

Sometimes, you cannot upgrade to a newer version, but you need a hotfix or whatever. It is crucial that you are able to checkout the exact version you use for your egg.

These are a lot of steps, and there are a lot of actions that can go wrong. Luckily, there is a way to automate it. zest.releaser provides scripts to release an egg, to check what has changed since the release and to check if the documentation has errors.

There once was a book on python. Among other things, it had a chapter on releasing an egg with sample code. The sample code was about a printer of nested lists. This resulted in a lot of packages to print out nested lists on pypi.

We will avoid this. Everybody, go to testpypi.python.org and create an account now.

Next, copy the pypirc_sample file to ~/.pypirc, modify it to contain your real username and password.

Now that we are prepared, let's install zest.releaser.

- lasttagdiff
- longtest
- prerelease
- release
- postrelease

Buildout II: Getting Ready for Deployment

The Starzel buildout

Have a look at the buildout some of the trainers use for their projects: https://github.com/starzel/buildout

It has some notable features:

- It extends to config- and version-files on github shared by all projects that use the same version of Plone:

  ```
  [buildout]
  extends =
      https://raw.githubusercontent.com/starzel/buildout/5.0.5/linkto/base.cfg
  ```

- It allows to update a project simply by changing the version it extends.
- It allows to update all projects of one version by changing remote files (very useful for HotFixes).
- It is minimal work to setup a new project.
- It has presets for development, testing, staging and production.
- It has all the nice development-helpers we use.

Another noteable buildout to look for inspiration:

- https://github.com/4teamwork/ftw-buildouts

A deployment setup

Deploying Plone and production-setups are outside the scope for this training. Please see http://docs.plone.org/manage/deploying/index.html

Other tools we use

There are plenty of tools that make developing and managing sites much easier. Here are only some of the ones you might want to check out:

- Fabric (managing sites)
- Sentry (error monitoring)
- Ansible (managing and provisioning machines)
- Greylog, ELK (logging)
- Nagios, Zabbix (server monitoring)
- jenkins, gitlab-ci, travis, drone.io (Continuous Integration)
- piwik (statistics)

- gitlab (code repo and code review)
- redmine, taiga, assembla (project-management and ticket-system)

Plone REST API

Get the code!

Get the code for this chapter (More info):

```
git checkout restapi
```

In this chapter, we will have a look at the relatively new addon plone.restapi. It provides a hypermedia API to access Plone content using REST (Representational State Transfer).

We will use `plone.restapi` to develop a small standalone 'single page app' targeted at mobile devices. We will present our users with a simple list of conference talks. We add lightning talks as a new type of talk. Users will be able to submit lightning talks e.g. using their mobile phone.

We have the following tasks:

- create a talk list view
- create a login screen and use JWT for authentication/authorization of requests
- let authenticated users submit lightning talks

Installing plone.restapi

We install `plone.restapi` like any other add-on package by adding it to `buildout.cfg` and then activating it in the *Add-ons* panel. This will automatically add and configure a new PAS plugin named *jwt_auth* used for JSON web token authentication.

Explore the API

Make sure you add some talks to the talks folder and then start exploring the API. We recommend using Postman or a similar tool, but you can also use requests in a Python virtual env. `plone.restapi` uses 'content negotiation' to determine whether a client wants a REST API response - if you set the `Accept` HTTP header to `application/json`, Plone will provide responses in JSON format. Some requests you could try:

```
GET /Plone/talks
Accept: application/json
```

```
POST /@login HTTP/1.1
Accept: application/json
Content-Type: application/json

{
    'login': 'admin',
    'password': 'admin',
}
```

Exercise

REST APIs use HTTP verbs for manipulating content. PATCH is used to update an existing resource. Add a new talk in Plone and then update it's title to match 'Foo 42' using the REST API (from Postman or requests).

Solution

We need to login to change content. Using JWT, we do so by POSTing credentials to the @login resource to obtain a JSON web token that we can subsequently use to authorize requests.

```
POST /@login HTTP/1.1
Accept: application/json
Content-Type: application/json

{
    'login': 'admin',
    'password': 'admin',
}
```

The response will look like this:

```
{
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
↪eyJmdWxsbmFtZSI6bnVsbCwic3ViIjoiYWRtaW4iLCJleHAiOjE0NzQ5MTU4Mzh9.s27se99V7leTVTo26N_
↪pbYskebR28W5NS87Fb7zowNk"
}
```

Using the requests library from Python, you would do:

```
>>> import requests
>>> response = requests.post('http://localhost:8080/Plone/@login',
...                          headers={'Accept': 'application/json', 'Content-Type':
↪'application/json'},
...                          data='{"login": "admin", "password": "admin"}')
>>> response.status_code
200
>>> response.json()
{'token': 'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
↪eyJmdWxsbmFtZSI6bnVsbCwic3ViIjoiYWRtaW4iLCJleHAiOjE0NzQ5MTYyNzR9.
↪zx8XJb6SCWB2taxyibLZ2461ibDloqU3QbWDkDzT8PY'}
>>>
```

Now we can change the talk title:

```
PATCH /Plone/talks/example-talk
Accept: application/json
Content-Type: application/json
Authentication: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
↪eyJmdWxsbmFtZSI6bnVsbCwic3ViIjoiYWRtaW4iLCJleHAiOjE0NzQ5MTYyNzR9.
↪zx8XJb6SCWB2taxyibLZ2461ibDloqU3QbWDkDzT8PY

{
    "@id": "http://localhost:8080/Plone/talks/example-talk",
    "title": "Foo 42"
}
```

Using requests again:

```
>>> requests.patch('http://localhost:8080/Plone/talks/example-talk',
...                headers={'Accept': 'application/json', 'Content-Type':
↪'application/json', 'Authorization': 'Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
↪eyJmdWxsbmFtZSI6bnVsbCwic3ViIjoiYWRtaW4iLCJleHAiOjE0NzQ5MTYyNzR9.
↪zx8XJb6SCWB2taxyibLZ2461ibDloqU3QbWDkDzT8PY'},
...                data='{"@id":"http://localhost:8080/Plone/talks/example-talk",
↪"title":"Foo 42"}')
<Response [204]>
```

Implementing the talklist

We will use Mobile Angular UI to develop our app. This is a relatively lightweight JavaScript framework for developing hybrid web apps built on top of AngularJS. There are a lot of other frameworks available (e.g. Ionic, OnsenUI, Sencha, ...), but most of them have more dependencies than *Mobile Angular UI*. For example, most of them require NodeJS as a development web server. Our focus is Plone and interacting with `plone.restapi`, and *Mobile Angular UI* perfectly suits our needs because it simply lets us use Plone as our development webserver.

To get started, we download the current master branch of Mobile Angular UI from Github, extract it and copy the `dist` folder into a new subdirectory of `browser` named `talklist`. So, assuming the current working directory is the buildout directory:

```
$ wget https://github.com/mcasimir/mobile-angular-ui/archive/master.zip
$ unzip master.zip
$ mkdir src/ploneconf.site/src/ploneconf/site/browser/talklist
$ cp -a mobile-angular-ui-master/dist src/ploneconf.site/src/ploneconf/site/browser/
↪talklist/
```

Then we add a new resource directory to `browser/configure.zcml`:

```
<browser:resourceDirectory
    name="talklist"
    directory="talklist"
    />
```

In the `browser/talklist` directory, we add an HTML page called `index.html`:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <base href="/Plone/++resource++talklist/" />
    <title>List Of Talks</title>
    <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1" />
    <meta name="apple-mobile-web-app-capable" content="yes" />
    <meta name="viewport" content="user-scalable=no, initial-scale=1.0, maximum-
↪scale=1.0, minimal-ui" />
    <meta name="apple-mobile-web-app-status-bar-style" content="yes" />
    <link rel="shortcut icon" href="/favicon.png" type="image/x-icon" />
    <link rel="stylesheet" href="dist/css/mobile-angular-ui-hover.min.css" />
    <link rel="stylesheet" href="dist/css/mobile-angular-ui-base.min.css" />
    <link rel="stylesheet" href="dist/css/mobile-angular-ui-desktop.min.css" />
  </head>
  <body
    ng-app="TalkListApp"
    ng-controller="MainController"
  >
  <h1>List of talks</h1>
  <div class="app">
   <!-- App Body -->
   <div class="app-body">
    <div class="scrollable-content section">
     <div class="panel-group"
      ui-shared-state="myAccordion"
      ui-default='2'>
      <div class="panel panel-default" ng-repeat="item in items">
       <div class="panel-heading" ui-set="{'myAccordion': item.pos}">
        <h4 class="panel-title">
         {{item.type}}: {{item.title}} by {{item.speaker}}
        </h4>
        <b>{{item.start}}</b>
       </div>
       <div ui-if="myAccordion == {{item.pos}}">
        <div class="panel-body">
         {{item.details}}
        </div>
       </div>
      </div>
     </div>
    </div>
   </div>
  </div><!-- ~ .app -->
  <script src="//ajax.googleapis.com/ajax/libs/angularjs/1.5.6/angular.min.js"></
↪script>
  <script src="//ajax.googleapis.com/ajax/libs/angularjs/1.5.6/angular-route.min.js
↪"></script>
  <script src="dist/js/mobile-angular-ui.min.js"></script>
  <script src="talklist.js"></script>
 </body>
</html>
```

Now you can point your browser to http://localhost:8080/Plone/++resource++talklist/index.html to see the result. So far, the page will simply display a list of published talks. But we also need some JavaScript that we put into a file named `talklist.js` in the same folder:

```
'use strict';

//
// module depends on mobile-angular-ui
//
var app = angular.module('TalkListApp', [
 'mobile-angular-ui',
]);


app.controller('MainController', function($rootScope, $scope, $http) {

 $scope.items = [];

 $scope.load_talks = function() {
  $http.get('/Plone/talks',
       {headers:{'Accept':'application/json'}}).
   success(function(data, status, headers, config) {
    $scope.items = [];
```

```
        // get the paths of the talks
        var paths = [];
        for (var i=0; i < data.items_total; i++) {
          paths.push(data.items[i]['@id'])
        }
        // next get details for each talk
        for (var i=0; i < paths.length; i++) {
          $http.get(paths[i],
                    {headers:{'Accept':'application/json'}}).
            success(function(talkdata, status, headers, config) {
              // this is an angular 'promise' - we cannot
              // rely on variables from an outer scope
              var path = talkdata['@id'];
              var talk = {
                'pos': paths.indexOf(path),
                'path': path,
                'title': talkdata.title,
                'type': talkdata.type_of_talk,
                'speaker': (talkdata.speaker != null) ? talkdata.speaker : talkdata.creators[0],
                'start': talkdata.start,
                'subjects': talkdata.subjects,
                'details': (talkdata.details != null) ? talkdata.details.data : talkdata.description
              }
              $scope.items.push(talk);

            }).
            error(function(talkdata, status, headers, config) {});
        }
      }).
    error(function(data, status, headers, config) {
      $scope.items = [];
    });
  };

  // initialize
  $scope.load_talks();
});
```

Submit lightning talks

We add a new type of talk: lightning talk. A lightning talk is a short presentation of up to 5 minutes duration that can cover just about any topic. The information we need to provide for lightning talks is far less than for the more formal types of talk. Often the information provided for lightning talks is restricted to the talk subject or title and the speaker name, but we allow for a short summary. Before they can submit a lightning talk, potential speakers will need to login and we will use their previously registered login name as the speaker's name to display in the talk list.

Before we can start to submit lightning talks using REST calls from our single page app, we have to adapt the talk schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<model xmlns="http://namespaces.plone.org/supermodel/schema"
   xmlns:form="http://namespaces.plone.org/supermodel/form"
   xmlns:i18n="http://xml.zope.org/namespaces/i18n"
   xmlns:lingua="http://namespaces.plone.org/supermodel/lingua"
   xmlns:marshal="http://namespaces.plone.org/supermodel/marshal"
   xmlns:security="http://namespaces.plone.org/supermodel/security"
   xmlns:users="http://namespaces.plone.org/supermodel/users">
  <schema>
    <field name="type_of_talk" type="zope.schema.Choice"
      form:widget="z3c.form.browser.radio.RadioFieldWidget">
      <description />
      <title>Type of talk</title>
      <values>
        <element>Talk</element>
        <element>Training</element>
        <element>Keynote</element>
        <element>Lightning Talk</element>
      </values>
    </field>
    <field name="details" type="plone.app.textfield.RichText">
      <description>Add a short description of the talk (max. 2000 characters)</
↪description>
      <max_length>2000</max_length>
      <title>Details</title>
      <required>False</required>
    </field>
    <field name="audience"
      type="zope.schema.Set"
      form:widget="z3c.form.browser.checkbox.CheckBoxFieldWidget">
      <description />
      <title>Audience</title>
      <value_type type="zope.schema.Choice">
        <values>
          <element>Beginner</element>
          <element>Advanced</element>
          <element>Professionals</element>
        </values>
      </value_type>
    </field>
    <field name="room"
      type="zope.schema.Choice"
      form:widget="z3c.form.browser.radio.RadioFieldWidget"
      security:write-permission="cmf.ReviewPortalContent">
      <description></description>
      <required>False</required>
      <title>Room</title>
      <vocabulary>ploneconf.site.vocabularies.Rooms</vocabulary>
    </field>
    <field name="speaker" type="zope.schema.TextLine">
      <description>Name (or names) of the speaker</description>
      <title>Speaker</title>
      <required>False</required>
    </field>
    <field name="email" type="plone.schema.email.Email">
      <description>Adress of the speaker</description>
      <title>Email</title>
      <required>False</required>
    </field>
    <field name="image" type="plone.namedfile.field.NamedBlobImage">
      <description />
      <required>False</required>
      <title>Image</title>
    </field>
    <field name="speaker_biography" type="plone.app.textfield.RichText">
      <description />
      <max_length>1000</max_length>
      <required>False</required>
      <title>Speaker Biography</title>
    </field>
  </schema>
</model>
```

Next, in our JavaScript code, we provide a method for logging in a user and another one to check whether the user has a valid JSON web token. We use the `localStorage` facility of the browser to store the token on the client.

```
...
app.controller('MainController', function($rootScope, $scope, $http) {
...
  $scope.login = function(login, passwd) {
    $http.post('/Plone/@login',
               {'login':login,
                'password':passwd},
               {headers:
                {'Content-type':'application/json',
                 'Accept':'application/json'}}).
      success(function(data, status, headers, config){
        localStorage.setItem('jwtoken', data.token);
      }).
      error(function(data, status, headers, config){
        alert('Could not log you in');
      });
  };

  $scope.is_logged_in = function() {
    // we assume the user is logged in when he has a JWT token (that is naive)
    return localStorage.getItem('jwtoken') != null;
  };
...
```

We continue with changes to `index.html` so that it uses the new methods. We provide a login form if the user doesn't have a valid JSON web token. Only authenticated users can see the rest of the page.

```
       <div class="app-body">

         <div class="scrollable">
           <div class="scrollable-content section" ng-if="! is_logged_in()">
             <form role="form" ng-submit='login(userid,passwd)'>
               <fieldset>
                 <legend>Login</legend>
                 <div class="form-group has-success has-feedback">
                   <label>Login</label>
                   <input type="text"
                     ng-model="userid"
                     class="form-control"
                     placeholder="Enter login">
                 </div>
                 <div class="form-group">
                   <label>Password</label>
                   <input type="password"
                     ng-model="passwd"
                     class="form-control"
                     placeholder="Password">
                 </div>
               </fieldset>
               <hr>
               <button class="btn btn-primary btn-block">
                 Login
               </button>
             </form>
           </div>

           <div class="scrollable-content section" ng-if="is_logged_in()">
             <div class="panel-group"
```

Last we have to add some code that allows authenticated users to submit a lightning talk. We add another javascript method first:

```
...
app.controller('MainController', function($rootScope, $scope, $http) {
...
  $scope.submit_talk = function(subject, summary) {
    $http.post('/Plone/talks',
               {'@type':'talk',
                'type_of_talk':'Lightning Talk',
                'audience':['Beginner','Advanced','Professionals'],
                'title':subject,
                'description':summary},
               {headers:
                {'Content-type':'application/json',
                 'Authorization': 'Bearer ' + localStorage.getItem('jwtoken'),
                 'Accept':'application/json'}}).
      success(function(data, status, headers, config){
        if(status==201) { // created
          $scope.load_talks();
        }
      }).
      error(function(data, status, headers, config){
        // according to docs, status can be 400 or 500
        // we check wether the token has expired - in this case,
        // we remove it from localStorage and disply the login page.
        // In all other cases, we display the message received
        // from Plone
        if ( (status == 400) && (data.type == 'ExpiredSignatureError') ) {
          localStorage.removeItem('jwtoken');
          location.reload();
        } else {
          // reason/error msg is contained in response body
          alert(data.message);
        }
      });
  };
...
```

Exercise

Rewrite the `load_talks()` javascript method so that it uses the portal search instead of `/Plone/talks`. Sort the list by date.

Solution

```
...
$scope.load_talks = function() {
  $http.get('/Plone/@search?portal_type=talk&sort_on=Date',
            {headers:{'Accept':'application/json'}}).
    success(function(data, status, headers, config) {
...
  });
```

The Future of Plone

- The Plone process, the various teams and and the Plone Community
- Plips: https://github.com/plone/Products.CMFPlone/issues?q=is%3Aopen+is%3Aissue+label%3A%2203+type%3A+feature+%28plip%29%22
- Plone 5.x
- Plone 6
- Plone 7 and beyond...
- Plone Roadmap: https://plone.org/roadmap

Optional

- zc3.form
- Portlets
- ZCA in depth
- ZODB
- RelStorage
- More and more complex fields
- Custom edit forms
- Users, authentication, member profiles, LDAP
- Caching (plone.app.caching)
- Migrations
- Asynchronous processing
- Talking with external APIs
- *Releasing Your Code*
- grok

- Professional Deployment

Please note that this document is *not complete* without the spoken word of a trainer. Even though we attempt to include the most important parts of what we teach in the narrative but reading it here can in no way be considered equal to attending a training.

Changelog

This changelog is only very rough. For the full changelog please refer to https://github.com/plone/training/commits/master

1.2.5 (unreleased)

- Fix typos, improve wording [svx]
- Clarify which template we're editing [djowett]
- Fix typos [tkimnguyen]
- Fix Sphinx warnings emitted on clean build [stevepiercy]
- Update README.rst to refer to how to build the docs locally. [stevepiercy]
- Add CONTRIBUTING.md [stevepiercy]
- Move PloneConf 2016 to Previous Trainings [stevepiercy]
- Update JavaScript training with latest exercises and documentation using collective.jstraining [vangheem]
- Update theming training to reflect the changes in bobtemplates.plone and general cleanup, also add refs to ttw training, remove usage of resource registry for theming [MrTango]
- Add solr training [tomgross]
- Rearrange structure so Mastering Plone now lives in it's own folder. [pbauer]
- Fix directions which led to duplicate resourcess being delivered Closes https://github.com/plone/training/issues/174 [davilima6]
- Plone Doc Style for Javascript part. [jensens]
- Add spell-checker, auto-build and travis-tests [svx]
- Use Plone 5 final and simplify vagrant-setup. [pbauer, fulv]
- Rewrite chapter on relations. [pbauer]
- Add a training on javasript. [frappel, thet]
- Add a training theming Plone 5. [MrTango, simahawk]
- A ton of fixes in the development-training in preparation to Ploneconf 2015 in Bucarest. [fulv]
- Update vagrant installation to include BIOS virtualization note. [lbrannon]
- Editing while reading. Edited Rapido chapter for language and formatting. [jean]
- Fix a couple of duplicate labels, unmatched literal ending, typo [jean]
- Fix code blocks that made Pygments lexer choke [jean]
- Fix some typos, clarify some examples [jean]
- Some exercises, draft of a new chapter on plone.restapi and some changes [tschorr]

- Exercises for behaviors_2, fix some emphasis [tschorr]
- Fix some typos, apply inline directives: file, Python domain, GUI, literals [jean]
- Add Plone 5 Workflow Training [calvinhp]
- Editing Dexterity chapters (typos, markup, some grammar) [jean]
- Fix reST lists [davisagli]
- Specify Python version for virtualenv using option, not command alias, as alias is not always present [jean]
- Fix duplicate labels [jean]
- Fix tests [gforcada]
- Review solr docs [gforcada]

1.2.4 (2014-10-03)

- Revision of part one for Ploneconf 2014, Bristol [smcmahon]
- Revised for Ploneconf 2014, Bristol [pbauer, gomez]
- Add first exercises and create css+js for expanding/collapsing the solutions [pbauer]
- Fix local build with the rtd-Theme [pbauer]
- Add Spanish translation. [macagua]
- Add support for translations on transifex [macagua]

1.2.3 (2014-07-11)

- Move sources to https://github.com/plone/training and render at https://plone-training.readthedocs.org/en/legacy/ [pbauer]
- Integrate with docs.plone.org and papyrus [do3cc]
- Change license to http://creativecommons.org/licenses/by/4.0/ [pbauer]
- Document how to contribute [pbauer]
- Update introduction [pbauer]

1.2.2 (2014-06-01)

- Fix all mistakes found during the training in May 2014 [pbauer]
- Move rst-files to https://github.com/starzel/training [pbauer]

1.2.1 (2014-05-30)

- Publish verbose version on http://starzel.github.io/training/index.html [pbauer]
- Add bash-command to copy the code from ploneconf.site_sneak to ploneconf.site for each chapter [pbauer]
- include vagrant-setup as zip-file [pbauer]
- several small bug fixes [pbauer]

1.2 (2014-05-23)

- Heavily expanded and rewritten for a training in Mai 2014 [pbauer, do3cc]
- remove grok [pbauer]
- use plone.app.contenttypes from the beginning [pbauer]
- use plone.api [pbauer]
- rewrite vagrant-setup [pbauer]
- drop use of plone.app.themeeditor [pbauer]
- add more chapters: Dexterity Types II: Growing up, User generated content, Programming Plone, Custom Search, Events, Using third-party behaviors, Dexterity Types III: Python, ... [pbauer, do3cc]

1.1 (October 2013)

- Revised and expanded for Ploneconf 2013, Brasilia [pbauer, do3cc]

1.0 (October, 2012)

- First version under the title 'Mastering Plone' for Ploneconf 2012, Arnhem [pbauer, do3cc]

0.2 October 2011

- Expanded as Plone-Tutorial for PyCon De 2011, Leipzig [pbauer]

0.1 (October 2009)

- Initial parts created for the Plone-Einsteigerkurs (http://www.plone.de/trainings/einsteiger-kurs/kursuebersicht) [pbauer]

Mastering Plone Theming

Basic: Customizing logo and CSS of default theme

In this section you will:

- Use the Site control panel to add a custom logo
- Customize the look of a Plone site by adjusting Less Variables
- Add a custom toolbar logo

Topics covered:

- The "Site" control panel
- The "Resource Registries" Control Panel
- Resource Registries > Development Mode

Customize logo

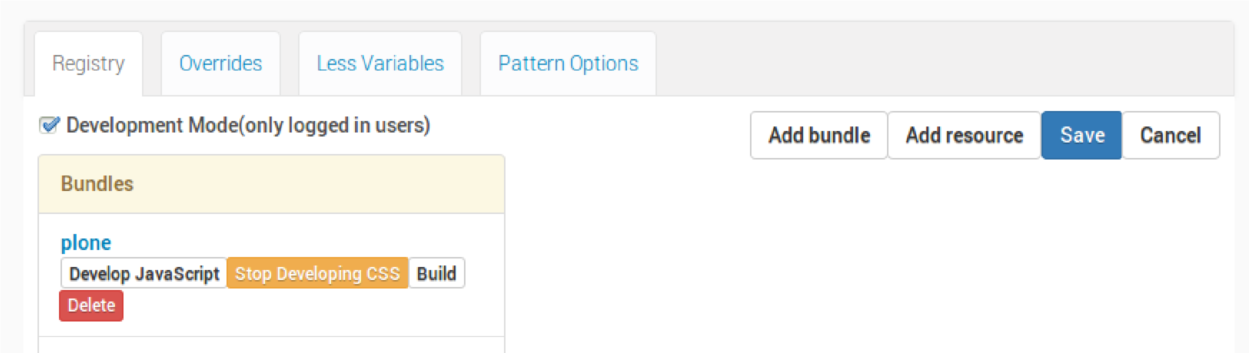
1. Go to the Plone Control Panel: *toolbar* → *admin* → *Site Setup*
2. Go to the “Site” control panel.
3. You will see this form:
4. You can now add / remove your custom logo

See the official docs.

Customize CSS/Less variables

1. Go back to the Control Panel.
2. Go to the *Resource Registries* control panel.
3. On the first tab: enable *Development Mode*.
4. In the “plone” bundle below, click on “develop CSS”.

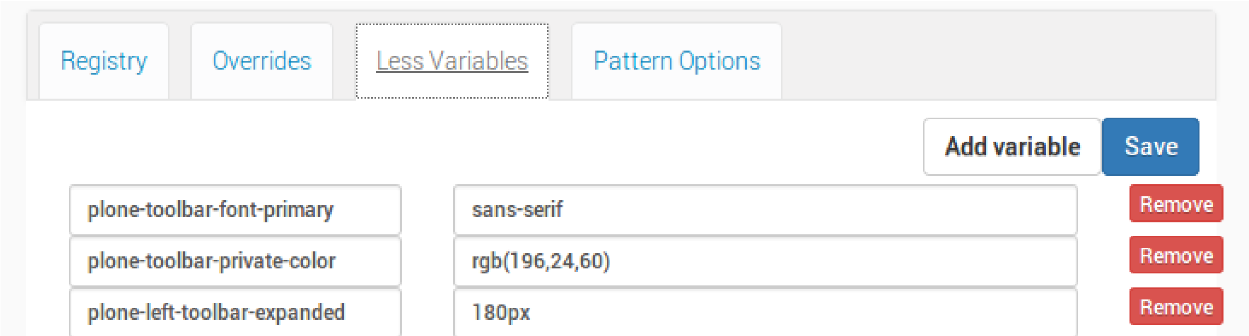
Your panel should now look like this:



☑ Development Mode(only logged in users)

Now we can play with some Less variables:

1. Go to the *Less Variables* tab.
2. Find the variable `plone-left-toolbar-expanded` and set it to 400px.



plone-toolbar-font-primary	sans-serif
plone-toolbar-private-color	rgb(196,24,60)
plone-left-toolbar-expanded	180px

3. Hit the *Save* button in the upper right and reload the page.
4. Click on the toolbar logo to expand the toolbar: voilá!

You can play around with some other variables, if you want.

Warning: "Development Mode" is really expensive for the browser. Depending on the browser and on the system, you might encounter extreme slowness while rendering the page. You could see an unthemed page for a while. Remember to switch it off as soon as you finished tweaking.

TTW Theming I: Introduction to Diazo Theming

In this section you will:

- Use the "Theming" control panel to make a copy of Plone's default theme (barceloneta)
- Customize a theme using Diazo rules
- Customize a theme by editing and compiling Less files

Topics covered:

- Diazo and plone.app.theming
- "Barceloneta" - The Default Plone Theme
- The "Theming tool"
- Building CSS in the "Theming tool"
- `<body>` element CSS classes
- Conditionally activating rules

Installation

We will use a Plone pre-configured Heroku instance.

Once deployed, create a Plone site.

Two approaches to theming

There are two main approaches to creating a custom theme:

1. Copying the default Barceloneta theme
2. Inheriting from the default Barceloneta theme.

In this section we'll look at the first approach, part II will explore the second approach.

What is Diazo?

`Diazo` is a theming engine used by Plone to make theming a site easier. At its core, a Diazo theme consists of an HTML page and `rules.xml` file containing directives.

Note: You can find extended information about Diazo and its integration package `plone.app.theming` in the official docs: Diazo docs and plone.app.theming docs.

Principles

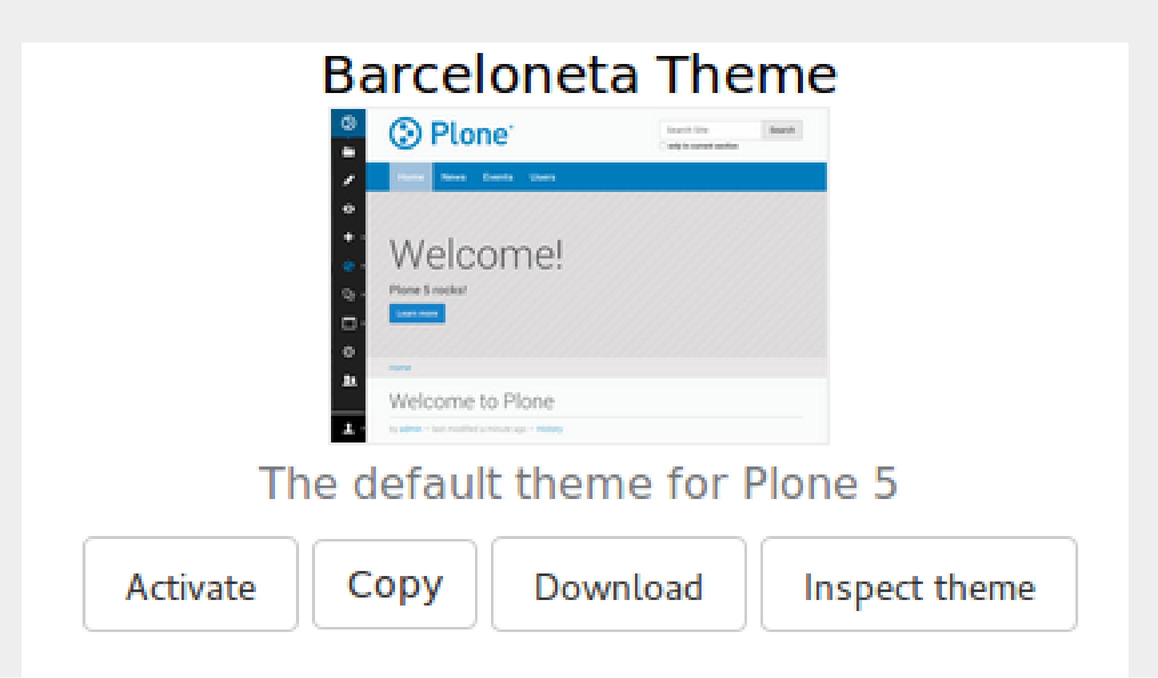
For this part of the training you just need to know the basic principles of a Diazo theme:

- Plone renders the content of the page;
- Diazo rules inject the content into any static theme;

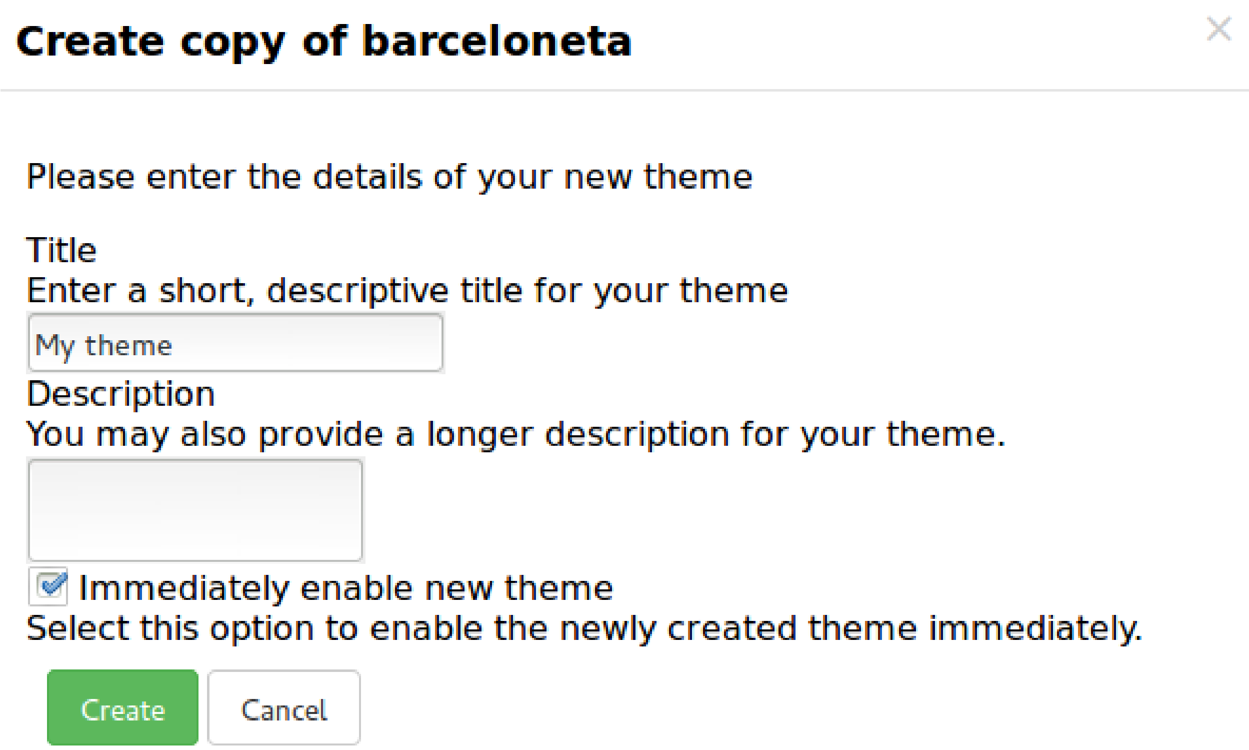
Copy barceloneta theme

To create our playground we will copy the existing Barceloneta theme.

1. go to the *Theming* control panel
2. you will see the available themes. In a bare new Plone site, you will see something like this:



3. click on the *Copy* button and get to the copy form
4. insert "My theme" as the name and activate it by default



Create copy of barceloneta

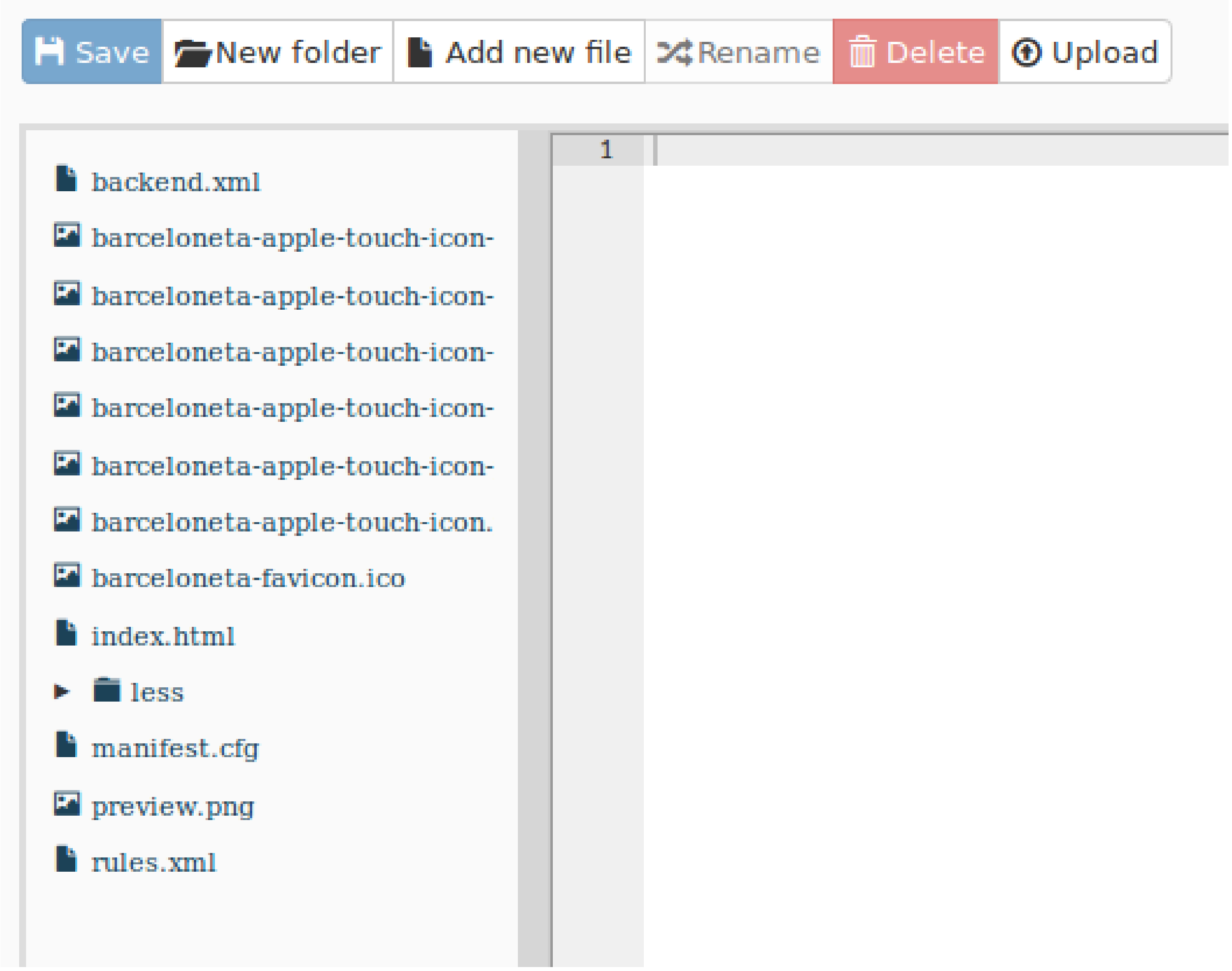
Please enter the details of your new theme

Title
Enter a short, descriptive title for your theme
My theme

Description
You may also provide a longer description for your theme.

☑ Immediately enable new theme
Select this option to enable the newly created theme immediately.

5. click on *Create* and you get redirected to your new theme's inspector:



backend.xml
barceloneta-apple-touch-icon-
barceloneta-apple-touch-icon-
barceloneta-apple-touch-icon-
barceloneta-apple-touch-icon-
barceloneta-apple-touch-icon-
barceloneta-apple-touch-icon.
barceloneta-favicon.ico
index.html
less
manifest.cfg
preview.png
rules.xml

Anatomy of a Diazo theme

The most important files:

- `manifest.cfg`: contains metadata about the theme (manifest reference);
- `rules.xml`: contains the theme rules (rules reference);
- `index.html`: the static HTML of the theme.

Exercise 1 - Inspecting the `manifest.cfg`

To better understand how your theme is arranged start by reading the `manifest.cfg` file.

In the theming tool, open the `manifest.cfg` spend a minute or two looking through it, then see if you can answer the questions below.

Where are the main rules located for your theme?

What property in the `manifest.cfg` file defines the source CSS/Less file used by the theme?

What do you think is the purpose of the `prefix` property?

Solution

The main rules are defined by the `rules` property (you could point this anywhere, however the accepted convention is to use a file named `rules.xml`.

The `development-css` property points at the main Less file, when compiled to CSS it is placed in the location defined by the `production-css` property.

The `prefix` property defines the default location to look for non prefixed files, for example if your prefix is set to `/++theme++mytheme` then a file like index.html will be expected at `/++theme++mytheme/index.html`

`<body>` CSS classes

As you browse a Plone site, Plone adds rich information about your current context. This information is represented as special classes in the `<body>` element. Information represented by the `<body>` classes includes:

- the current user role, and permissions,
- the current content-type and its template,
- the site section and sub section,
- the current subsite (if any),
- whether this is a frontend view,
- whether icons are enabled.

`<body>` classes for an anonymous visitor

Below you can see an example of the body classes for a page named “front-page”, located in the root of a typical Plone site called “acme”:

```
<body class="template-document_view
             portaltype-document
             site-acme
             section-front-page
             icons-on
             thumbs-on
             frontend
             viewpermission-view
             userrole-anonymous">
```

`<body>` classes for a manager

And here is what the classes for the same page look like when viewed by a manager that has logged in:

```
<body class="template-document_view
             portaltype-document
             site-acme
             section-front-page
             icons-on
             thumbs-on
             frontend
             viewpermission-view
             userrole-member
             userrole-manager
             userrole-authenticated
             plone-toolbar-left
             plone-toolbar-expanded
             plone-toolbar-left-expanded">
```

Notice the addition of `userrole-manager`.

Exercise 2 - Discussion about the `<body>` classes

Look back at the `<body>` classes for a manager then see if you can answer the following questions.

1. What other roles does the manager have?
2. Can you see other differences?
3. What do you think the `plone-toolbar-expanded` class does?

Solution

The manager also has the role "member" and "authenticated"

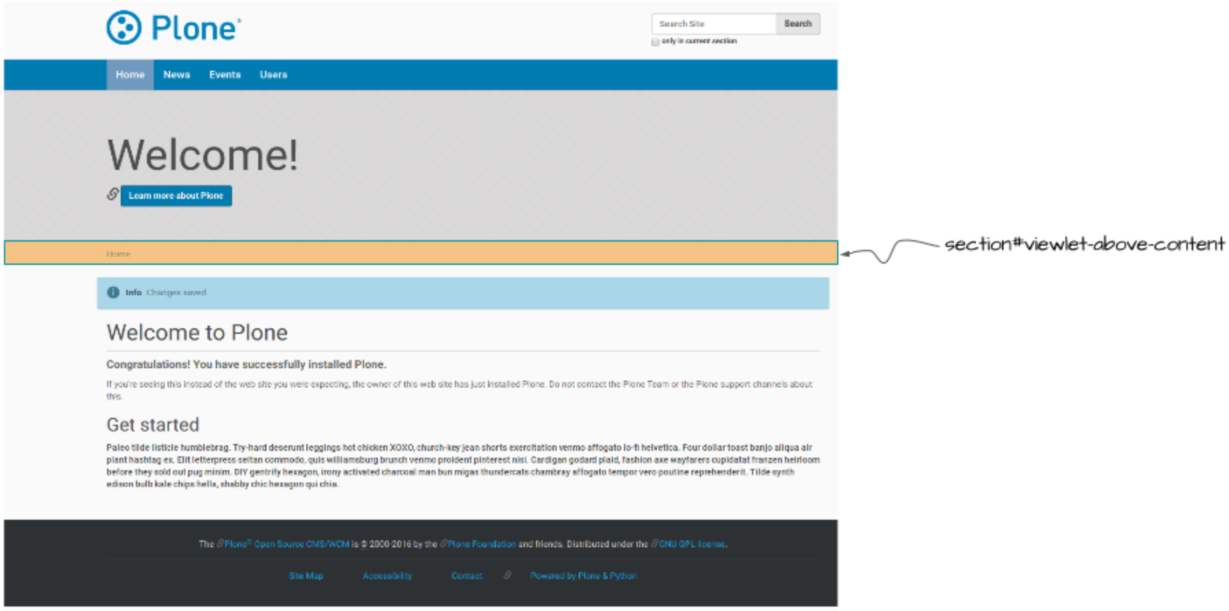
There are `plone-toolbar` classes added to the `<body>` element, these control the display of the toolbar

The `plone-toolbar-expanded` class is used to control styles used by the expanded version of the toolbar.

Custom rules

Let's open `rules.xml`. You will see all the rules that are used in the Barceloneta theme right now. For the time being let's concentrate on how to hack these rules.

Conditionally showing content



Changes saved

Welcome to Plone

Congratulations! You have successfully installed Plone.

If you're seeing this instead of the web site you were expecting, the owner of this web site has just installed Plone. Do not contact the Plone Team or the Plone support channels about this.

Get started

Paleo tilde listicle humblebrag. Try-hard deserunt leggings hot chicken XOXO, church-key jean shorts exercitation venmo affogato lo-fi helvetica. Four dollar toast banjo aliqua air plant hashtag ex. Elit letterpress seitan commodo, quis williamsburg brunch venmo proident pinterest nisi. Cardigan godard plaid, fashion axe wayfarers cupidatat franzen heirloom before they sold out pug minim. DIY gentrify hexagon, irony activated charcoal man bun migas thundercats chambray affogato tempor vero poutine reprehenderit. Tilde synth edison bulb kale chips hella, shabby chic hexagon qui chia.

The Plone® Open Source CMS/WCM is © 2000-2016 by the Plone Foundation and friends. Distributed under the GNU GPL license.

Suppose that we want to make the "above content" block (the one that contains breadcrumbs) conditional, and show it only for authenticated users.

In the `rules.xml` find this line:

```
<replace css:content="#viewlet-above-content" css:theme="#above-content" />
```

This rule states that the element that comes from the content (Plone) with the id `#viewlet-above-content` must replace the element with the id `#above-content` in the static theme.

We want to hide it for anonymous users (hint: We'll use the `<body>` classses discussed above).

The class we are looking for is `userrole-authenticated`. Add another property to the rule so that we produce this code:

```
<replace
    css:if-content="body.userrole-authenticated"
    css:content="#viewlet-above-content"
    css:theme="#above-content" />
```

The attribute `css:if-content` allows us to put a condition on the rules based on a CSS selector that acts on the content. In this way the rule will be applied only if the body element has the class `.userrole-authenticated`.

We will learn more about Diazo rules in *TTW Theming II: Creating a custom theme based on Barceloneta*.

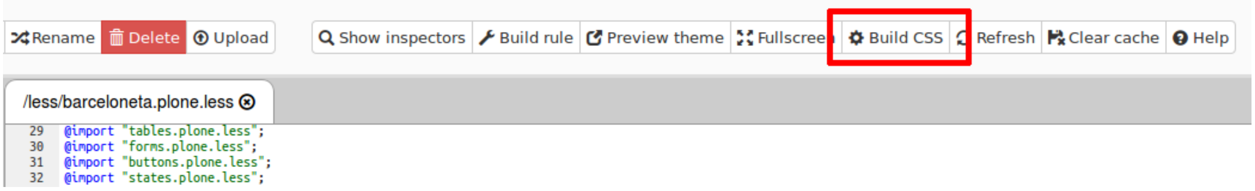
Customize CSS

1. from theming tool open the file `less/barceloneta.plone.less`, that is the main Less file as specified in the manifest;
2. add your own customization at the bottom, like:

```
body{ background-color: red; font-size: 18px ;};
```

Note: Normally you would place this in a separate file to keep the main one clean but for this example it is enough.

3. push the buttons *Save* and *Build CSS*



```
@import "tables.plone.less";
@import "forms.plone.less";
@import "buttons.plone.less";
@import "states.plone.less";
```

4. go back to the Plone site and reload the page: voilá!

Warning: At the moment you need to "Build CSS" from the main file, the one declared in the manifest (in this case `less/barceloneta.plone.less`). So, whatever Less file you edit, go back to the main one to compile. This behavior will be improved but for now, just remember this simple rule ;)

TTW Theming II: Creating a custom theme based on Barceloneta

In this section you will:

- Create a theme by inheriting from the Barceloneta theme.
- Using the `manifest.cfg`, register a production CSS file.
- Use an XInclude to incorporate rules from the Barceloneta theme.
- Use `?diazo.off=1` to view unthemed versions.
- Use conditional rules to have a different backend theme from the anonymous visitors theme.

Topics covered:

- Inheriting from Barceloneta.
- Diazo rule directives and attributes.
- Viewing the unthemed version of a Plone item.
- Creating a visitor-only theme.

Inheriting from Barceloneta

Key Ideas

When inheriting from the Barceloneta theme keep the following in mind:

- The theme provides styles and assets used by Plone's backend tools.
- Inheritance involves including the Barceloneta `rules.xml` (`++theme++barceloneta/rules.xml`) and styles.
- The prefix/unique path to the Barceloneta theme is `++theme++barceloneta`.
- It is necessary to include a copy of Barceloneta's `index.html` in the root of your custom theme.

- The three key files involved are `manifest.cfg`, `rules.xml` and a Less file defined in the manifest which we will call `styles.less`.
- Use "Build CSS" to generate a CSS file from your custom Less file.

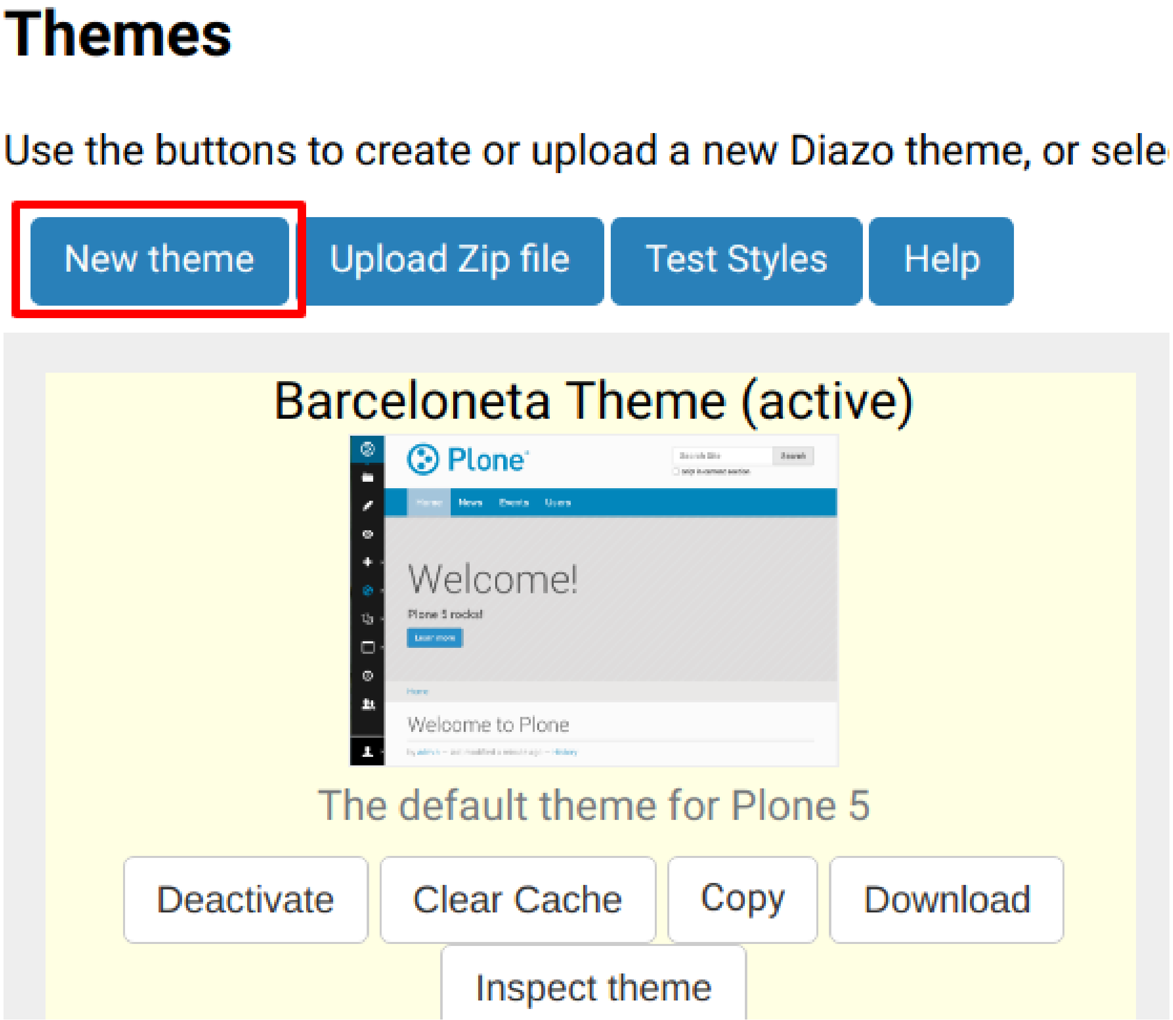
Copying Barceloneta makes your theme heavier and will likely make upgrading more difficult.

The Barceloneta theme provides many assets used by Plone's utilities that you do not need to duplicate. Additionally new releases of the theme may introduce optimizations or bug fixes. By referencing the Barceloneta rules and styles, instead of copying them, you automatically benefit from any updates to the Barceloneta theme while also keeping your custom theme relatively small.

Exercise 1 - Create a new theme that inherits from Barceloneta

In this exercise we will create a new theme that inherits the Barceloneta rules and styles.

1. Create a new theme



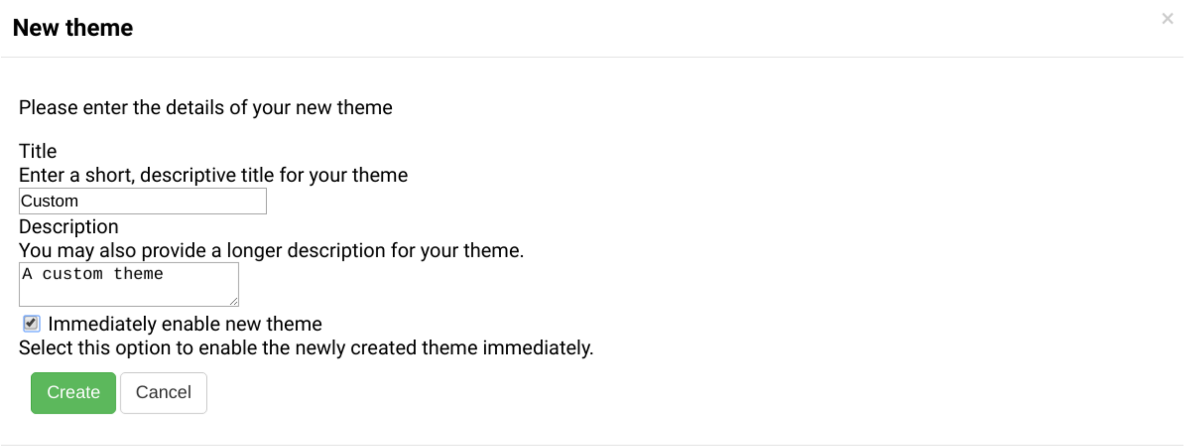
Themes

Use the buttons to create or upload a new Diazo theme, or sele

Barceloneta Theme (active)

The default theme for Plone 5

and name it "Custom"



New theme

Please enter the details of your new theme

Title
Enter a short, descriptive title for your theme
Custom

Description
You may also provide a longer description for your theme.
A custom theme

☑ Immediately enable new theme
Select this option to enable the newly created theme immediately.

2. In the theming editor, ensure that your new theme contains the files `manifest.cfg`, `rules.xml`, `index.html` (from Barceloneta) and `styles.less`.

- `manifest.cfg`, declaring your theme:

```
[theme]
title = mytheme
description =
development-css = ++theme++custom/styles.less
production-css = ++theme++custom/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 generate the `styles.css` file using `styles.less` and the "Build CSS" button.

Your theme is ready.

Diazo rule directives and attributes

The Diazo rules file is an XML document containing rules to specify where the content elements (title, footer, main text, etc.) will be located in the targeted theme page. The rules are created using *rule directives* which have *attributes*; attribute values are either CSS expressions or XPath expressions.

CSS selector based attributes

It is generally recommended that you use CSS3 selectors to target elements in your content or theme. The CSS3 selectors used by Diazo directives are listed below:

`css:theme` Used to select target elements from the theme using CSS3 selectors.

`css:content` Used to specify the element that should be taken from the content.

`css:theme-children` Used to select the children of matching elements.

`css:content-children` Used to identify the children of an element that will be used.

XPath selector based attributes

Depending on complexity of the required selector it is sometimes necessary or more convenient to use XPath selectors instead of CSS selectors. XPath selectors use the unprefixed attributes `theme` and `content`. The common XPath selector attributes include:

`theme` Used to select target elements from the theme using XPath selectors.

`content` Used to specify the element that should be taken from the content using XPath selectors.

`theme-children` Used to select the children of matching elements using XPath selectors.

`content-children` Used to identify the children of an element that will be used using XPath selectors.

You can also create conditions about the current path using `if-path`.

Note: For a more comprehensive overview of all the Diazo rule directives and related attributes see: http://docs.diazo.org/en/latest/basic.html#rule-directives

Viewing the unthemed Plone site

When you create your Diazo rules, it is important to know how the content Diazo is receiving from Plone is structured. In order to see a "non-diazoed" version page, just add `?diazo.off=1` at the end of its URL.

Exercise 2 - Viewing the unthemed site

1. Use `diazo.off=1` to view the unthemed version of your site.
2. Using your browser's inspector, find out the location/name of some of Plone's elements. Then try to answer the following:

   What do you think is the difference between "content-core" and "content"? There are several viewlets, how many do you count? Can you identify any portlets, what do you think they are for?

   Solution

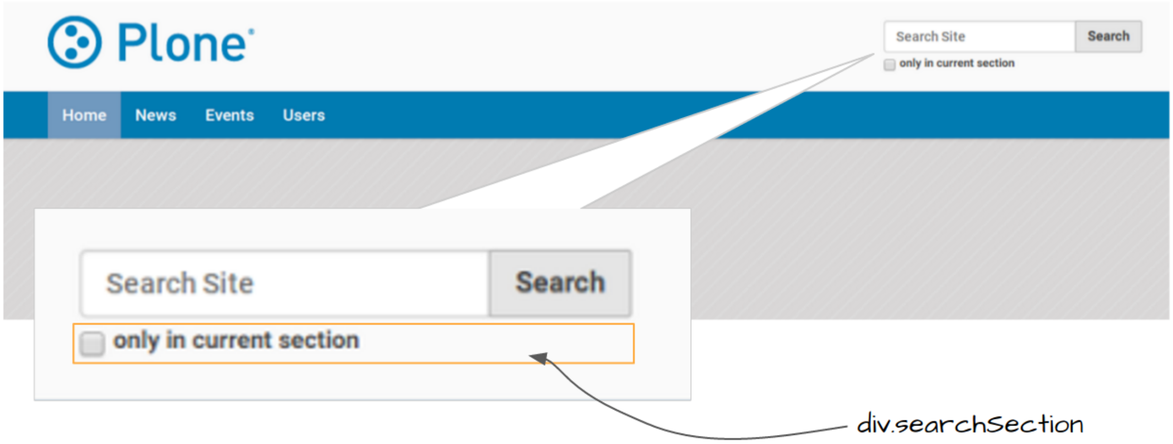
   The "content-core" does not include the "title" and "description" while the "content" combines the "title", "description" and "content-core".

   Out of the box there are six viewlets (`viewlet-above-content`, `viewlet-above-content-title` `viewlet-below-content-title`, `viewlet-above-content-body`, `viewlet-below-content-body`, `viewlet-below-content`).

   There are a few *footer* portlets which construct the footer of the site.

Exercise 3 - the `<drop>` directives

1. Add a rule that drops the "search section" checkbox from the search box. See the diagram below:



Conditional attributes

The following attributes can be used to conditionally activate a directive.

`css:if-content` Defines a CSS3 expression: if there is an element in the *content* that matches the expression then activate the directive.

`css:if-theme` Defines a CSS3 expression: if there is an element in the *theme* that matches the expression then activate the directive.

`if-content` Defines an XPath expression: if there is an element in the *content* that matches the expression then activate the directive.

if-theme Defines an XPath expression: if there is an element in the *theme* that matches the expression then activate the directive.

if-path Conditionally activate the current directive based on the current path.

Note: In a previous chapter we discussed the Plone `<body>` element and how to take advantage of the custom CSS classes associated with it. We were introduced to the attribute `css:if-content`. Remember that we are able to determine a lot of context related information from the classes, such as:

```
- the current user role, and its permissions,
- the current content-type and its template,
- the site section and sub section,
- the current subsite (if any).
```

Here is an example

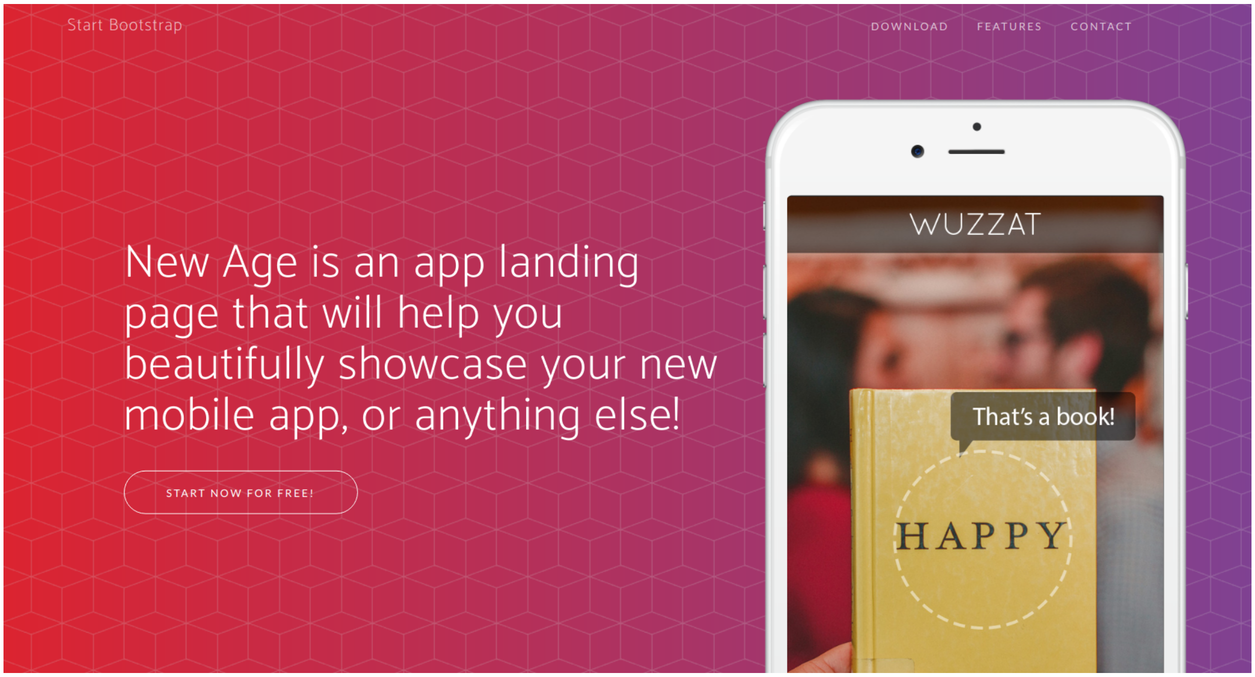
```
<body class="template-summary_view
             portaltype-collection
             site-Plone
             section-news
             subsection-aggregator
             icons-on
             thumbs-on
             frontend
             viewpermission-view
             userrole-manager
             userrole-authenticated
             userrole-owner
             plone-toolbar-left
             plone-toolbar-expanded
             plone-toolbar-left-expanded
             pat-plone
             patterns-loaded">
```

Converting an existing HTML template into an theme

In the Plone “universe” it is not uncommon to convert an existing HTML template into a Diazo theme. Just ensure that when you zip up the source theme that there is a single folder in the root of the zip file. We will explore this in more detail in the next exercise.

Exercise 4 - Convert a HTML template into a Diazo theme

In this exercise we will walk through the process of converting an existing free HTML theme into a Diazo-based Plone theme.



Start Bootstrap

DOWNLOAD FEATURES CONTACT

New Age is an app landing page that will help you beautifully showcase your new mobile app, or anything else!

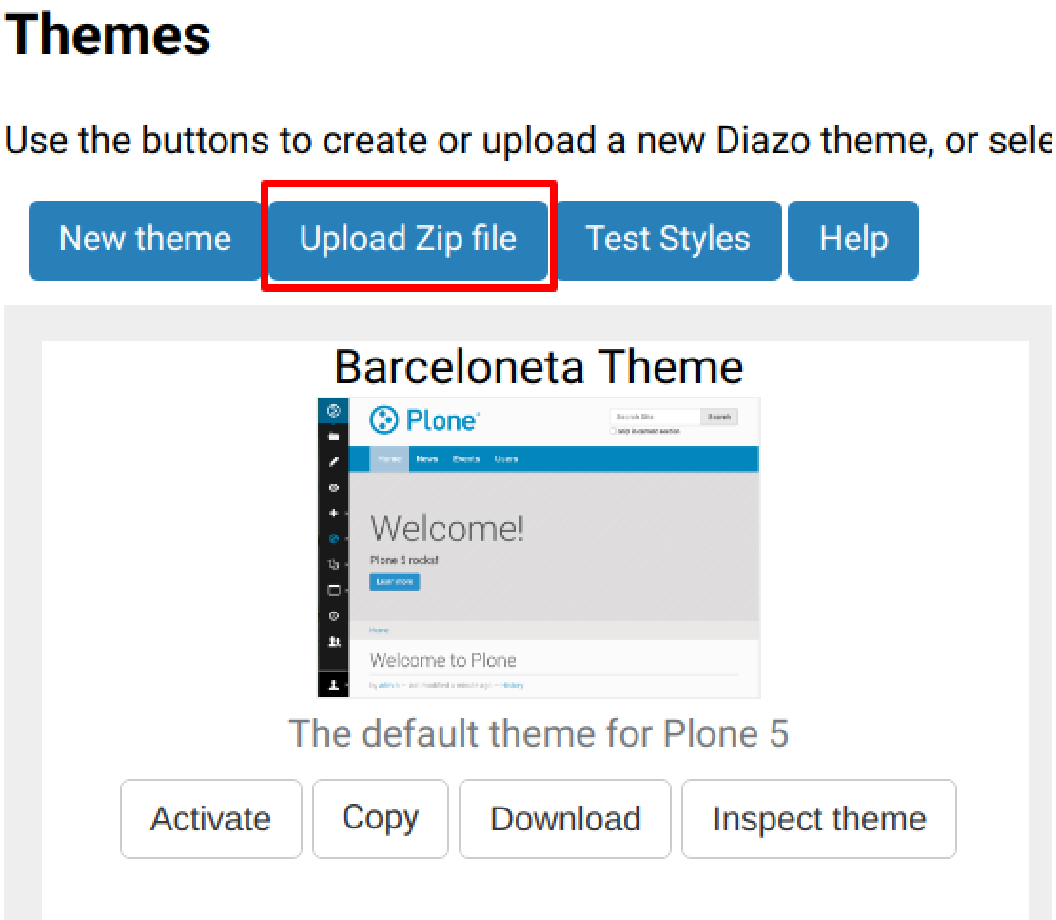
START NOW FOR FREE!

We've selected the free New Age Bootstrap theme. The theme is already packaged in a manner that will work with the theming tool.

Note: When being distributed, Plone themes are packaged as zip files. A theme should be structured such that there is only one top level directory in the root of the zip file. By convention the directory should contain your `index.html` and supporting files, the supporting files (CSS, javascript and other files) may be in subdirectories.

1. To get started download a copy of the New Age theme as a zip file. Then upload it to the theme controlpanel.

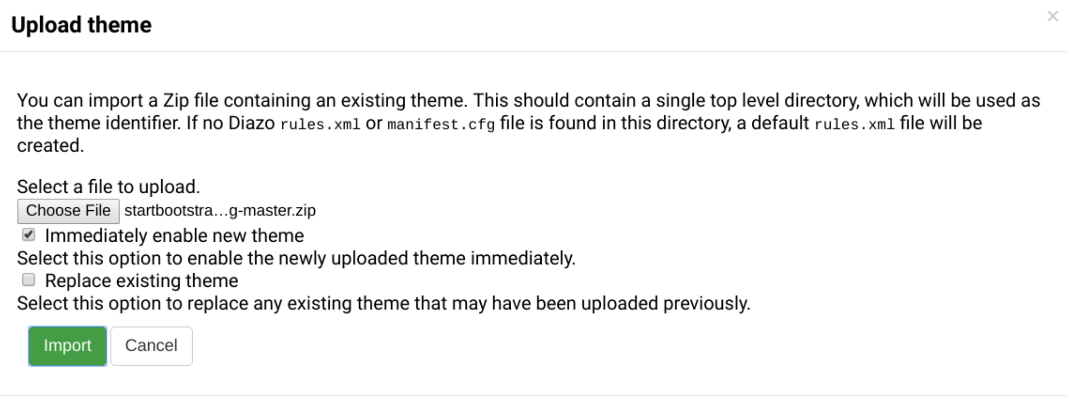
   Hint: This is a generic theme, it does not provide the Plone/Diazo specific `rules.xml` or `manifest.cfg` file. When you upload the zip file the theming tool generates a `rules.xml`. In the next steps you will add additional files including a `manifest.cfg` (perhaps in the future the manifest.cfg will also be generated for you).



Themes

Use the buttons to create or upload a new Diazo theme, or sele

Select the downloaded zip file.



Upload theme

You can import a Zip file containing an existing theme. This should contain a single top level directory, which will be used as the theme identifier. If no Diazo `rules.xml` or `manifest.cfg` file is found in this directory, a default `rules.xml` file will be created.

Select a file to upload.

Choose File startbootstra...g-master.zip

☑ Immediately enable new theme

Select this option to enable the newly uploaded theme immediately.

☐ Replace existing theme

Select this option to replace any existing theme that may have been uploaded previously.

2. Add a `styles.less` file and import the Barceloneta styles.
3. Add a `manifest.cfg` file, set `production-css` equal to `styles.css`

Note: Clean Blog is a free Bootstrap theme, the latest version is available on github https://github.com/BlackrockDigital/startbootstrap-clean-blog

Hint: You can identify the theme path by reading your browser's address bar when your theme is open in the theming tool. You'll need to include the proper theme path in your `manifest.cfg`, in this case it will most likely be something like `++theme++startbootstrap-new-age-gh-pages`

[theme] title = New Age prefix = ++theme++startbootstrap-new-age-gh-pages/ production-css = ++theme++startbootstrap-new-age-gh-pages/styles.css

4. Add rules to include the Barceloneta backend utilities

```
   <?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" />
```

5. Add rules to include content, add site structure, drop unneeded elements, customize the menu.

Warning: Look out for inline styles in this theme (i.e. the use of the `style` attribute on a tag). This is especially problematic with background images set with relative paths. The two issues that result are:

- the relative path does not translate properly in the context of the theme;
- it can be tricky to dynamically replace background images provided by inline styles.

Creating a visitor-only theme - conditionally enabling Barceloneta

Sometimes it is more convenient for your website administrators to use Barceloneta, Plone's default theme. Other visitors would see a completely different layout provided by your custom theme. To achieve this you will need to associate your visitor theme rules with an expression like `css:if-content="body.userrole-anonymous"`. For rules that will affect logged-in users you can use the expression `css:if-content="body.:not(userrole-anonymous)"`.

Once you've combined the expressions above with the right Diazo rules you will be able to present an anonymous visitor with a specific HTML theme while presenting the Barceloneta theme to logged-in users.

Warning: The Barceloneta `++theme++barceloneta/rules.xml` expects the Barceloneta `index.html` to reside locally in your current theme. To avoid conflict and to accomodate the inherited Barceloneta, ensure that your theme file has a different name such as `front.html`.

Exercise 5 - Convert the theme to be a visitor-only theme

In this exercise we will alter our theme from the previous exercise to make it into a visitor-only theme.

1. Update the `rules.xml` file to include Barceloneta rules.

Hint: Use `<xi:include href="++theme++barceloneta/rules.xml" />`

2. Add conditional rules to `rules.xml` so that the new theme is only shown to anonymous users, rename the theme's `index.html` to `front.html` and add a copy of the Barceloneta `index.html`.

Hint: Copy the contents of the Barceloneta `index.html` file then add it to the theme as the new `index.html` file.

Change `rules.xml` to look similar to this:

```
<?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">

  <notheme css:if-not-content="#visual-portal-wrapper" />

  <rules css:if-content="body:not(.userrole-anonymous)">
    <!-- Import Barceloneta rules -->
    <xi:include href="++theme++barceloneta/rules.xml" />
  </rules>

  <rules css:if-content="body.userrole-anonymous">
    <theme href="front.html" />
    <replace css:theme-children=".intro header h2" css:content-
↪children=".documentFirstHeading" />
    <replace css:theme-children=".summary" css:content-children=".
↪documentDescription" />
    <replace css:theme-children=".preamble" css:content-children="
↪#content-core" />
  </rules>
</rules>
```

Make it reproducible: static theme

You just created your shiny brand new theme TTW.

Note: For more TTW configuring and customizing options, see *"Through-the-web" Plone customization*

Now, let's see how you can reuse it in another Plone site.

1. go back to "Theming" control panel
2. click the "Download" button in the box of your theme
3. the browser will download a zip file
4. go to the ZMI root
5. create a new plone site
6. go to "Theming" control panel

7. click on "Upload zip file" and select your theme
8. tick "Immediately enable new theme" and click on "import"
9. go back to your plone site: voilá!

Create a Plone Theme python package

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 and you are also limited in what you can do that way.

Even if `plone.app.theming` allows importing and exporting of a Diazo theme as a ZIP archive, it might be preferable to manage your theme as an actual Plone product.

One of the most obvious reasons is that it will allow you to override Plone elements that are not accessible via pure Diazo features (such as overloading content view templates, viewlets, configuration settings, etc.).

Preparing your setup

Install npm

If you don't have already installed `npm` on your system please do it. Npm comes with nodejs, we just need to install `npm`. On Debian/Ubuntu for example you can do this with apt:

```
$ sudo apt install -y npm
```

If you need a newer version of `npm` just update your version with `npm` it self:

Installing Grunt

We also need to install `grunt-cli` globaly. If you already have it, you can skip this step.

```
$ npm install -g grunt-cli
```

Note: If you get an error with node on Debian/Ubuntu, please check if you already have /usr/bin/node if not create a symlink like: `ln -s /usr/bin/nodejs /usr/bin/node`.

virtualenv and mr.bob

First let's create a Python virtualenv:

```
$ virtualenv mrbobvenv
```

Then we enable the virtualenv:

```
$ source mrbobvenv/bin/activate
(mrbobvenv):~$
```

Create a product to handle your Diazo theme

To create a Plone 5 theme skeleton, you will use mr.bob's templates for Plone.

Install mr.bob and bobtemplates.plone

To install `mr.bob`, you can use **pip**:

```
(mrbobvenv):$ pip install mr.bob
```

and to install the required bobtemplates for Plone, do:

```
(mrbobvenv):$ pip install git+https://github.com/plone/bobtemplates.plone.git@1.0.5_
↪boston --upgrade
```

after bobtemplates.plone >= 1.0.5 is released do:

```
(mrbobvenv):$ pip install bobtemplates.plone
```

Create a Plone 5 theme product skeleton with **mrbob**:

```
(mrbobvenv):$ mrbob -O plonetheme.tango bobtemplates:plone_addon
```

It will ask you some question:

```
--> 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 however you like:

```
--> Theme name [Tango]: tango.de

--> 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 [5.0.5]:

Generated file structure at /home/maik/develop/plone/plonetheme.tango
```

Now you have a new Python package in your current folder:

```
(mrbobvenv):~/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
```

Deactivate mrbob virtualenv:

```
(mrbobvenv):~/develop/plone/plonetheme.tango$ deactivate
```

Install Buildout and boostrap your development environment

You can install Buildout globally or on a virtualenv. To install zc.buildout globally:

```
$ sudo pip install zc.buildout
```

```
$ buildout bootstrap
```

Now you have everything in place and you can run buildout:

```
$ ./bin/buildout
```

This will create the whole development environment for your package:

```
$ ls bin
addchangelogentry                code-analysis-jscs      grunt-task-compile ␣
 ↪pildriver.py  ride
buildout                         code-analysis-jshint    i18ndude            pilfile.
 ↪py    robot
bumpversion                      code-analysis-zptlint   instance            pilfont.
 ↪py    robot-debug
check-manifest                   createfontdatachunk.py  lasttagdiff         pilprint.
 ↪py   robot-server
code-analysis                    develop                 lasttaglog          player.
 ↪py     test
code-analysis-check-manifest     enhancer.py             libdoc             ␣
 ↪postrelease   thresholder.py
code-analysis-clean-lines        explode.py              longtest           ␣
 ↪prerelease    viewer.py
code-analysis-csslint            flake8                  npm-install         pybabel
code-analysis-find-untranslated  fullrelease             painter.py          pybot
code-analysis-flake8             gifmaker.py             pilconvert.py       release
```

Inspect your package source

Your package source code is in the `src` folder:

```
$ tree src/plonetheme/tango/
-- browser
|   -- configure.zcml
|   -- __init__.py
|   -- overrides
|   -- static
-- configure.zcml
-- __init__.py
-- interfaces.py
-- locales
|   -- plonetheme.tango.pot
|   -- update.sh
-- profiles
|   -- default
|   |   -- browserlayer.xml
|   |   -- metadata.xml
|   |   -- registry.xml
|   |   -- theme.xml
|   -- uninstall
|       -- browserlayer.xml
|       -- theme.xml
-- setuphandlers.py
-- testing.py
-- tests
|   -- __init__.py
|   -- robot
|   |   -- test_example.robot
|   -- test_robot.py
|   -- test_setup.py
-- theme
    -- backend.xml
    -- barceloneta
    |   -- 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.gridframework.plone.less
    |       -- mixin.grid.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
    |       -- scaffolding.plone.less
    |       -- search.plone.less
    |       -- sitemap.plone.less
    |       -- sitenav.plone.less
    |       -- sortable.plone.less
    |       -- states.plone.less
    |       -- tablesorter.plone.less
    |       -- tables.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
    -- barceloneta-apple-touch-icon-114x114-precomposed.png
    -- barceloneta-apple-touch-icon-144x144-precomposed.png
    -- barceloneta-apple-touch-icon-57x57-precomposed.png
    -- barceloneta-apple-touch-icon-72x72-precomposed.png
    -- barceloneta-apple-touch-icon.png
    -- barceloneta-apple-touch-icon-precomposed.png
    -- barceloneta-favicon.ico
    -- index.html
    -- less
    |   -- custom.less
    |   -- plone.toolbar.vars.less
    |   -- roboto
    |   |   -- LICENSE.txt
    |   |   -- RobotoCondensed-Light.eot
    |   |   -- RobotoCondensed-LightItalic.eot
    |   |   -- RobotoCondensed-LightItalic.svg
    |   |   -- RobotoCondensed-LightItalic.ttf
    |   |   -- RobotoCondensed-LightItalic.woff
    |   |   -- RobotoCondensed-Light.svg
    |   |   -- RobotoCondensed-Light.ttf
    |   |   -- RobotoCondensed-Light.woff
    |   |   -- Roboto-Light.eot
    |   |   -- Roboto-LightItalic.eot
    |   |   -- Roboto-LightItalic.svg
    |   |   -- Roboto-LightItalic.ttf
    |   |   -- Roboto-LightItalic.woff
    |   |   -- Roboto-Light.svg
    |   |   -- Roboto-Light.ttf
    |   |   -- Roboto-Light.woff
    |   |   -- Roboto-Medium.eot
    |   |   -- Roboto-MediumItalic.eot
    |   |   -- Roboto-MediumItalic.svg
    |   |   -- Roboto-MediumItalic.ttf
    |   |   -- Roboto-MediumItalic.woff
    |   |   -- Roboto-Medium.svg
    |   |   -- Roboto-Medium.ttf
    |   |   -- Roboto-Medium.woff
    |   |   -- Roboto-Regular.eot
    |   |   -- Roboto-Regular.svg
    |   |   -- Roboto-Regular.ttf
    |   |   -- Roboto-Regular.woff
    |   |   -- Roboto-Thin.eot
    |   |   -- Roboto-ThinItalic.eot
    |   |   -- Roboto-ThinItalic.svg
    |   |   -- Roboto-ThinItalic.ttf
    |   |   -- Roboto-ThinItalic.woff
    |   |   -- Roboto-Thin.svg
```

```
    |   |   -- Roboto-Thin.ttf
    |   |   -- Roboto-Thin.woff
    |   -- theme.less
    |   -- theme.local.less
    -- manifest.cfg
    -- package.json
    -- preview.png
    -- rules.xml
    -- template-overrides
    -- tinymce-templates
    |   -- image-grid-2x2.html
    -- views
        -- slider-images.pt.example
```

As you can see, the package already contains a Diazo theme including Barceloneta resources:

```
$ tree -L 2 src/plonetheme/tango/theme/
src/plonetheme/tango/theme/
-- backend.xml
-- barceloneta
|   -- less
-- barceloneta-apple-touch-icon-114x114-precomposed.png
-- barceloneta-apple-touch-icon-144x144-precomposed.png
-- barceloneta-apple-touch-icon-57x57-precomposed.png
-- barceloneta-apple-touch-icon-72x72-precomposed.png
-- barceloneta-apple-touch-icon.png
-- barceloneta-apple-touch-icon-precomposed.png
-- barceloneta-favicon.ico
-- HOWTO_DEVELOP.rst
-- index.html
-- less
|   -- custom.less
|   -- plone.toolbar.vars.less
|   -- roboto
|   -- theme-compiled.css
|   -- theme-compiled.css.map
|   -- theme.less
|   -- theme.local.less
-- manifest.cfg
-- node_modules
|   -- bootstrap
-- package.json
-- preview.png
-- rules.xml
-- template-overrides
-- tinymce-templates
|   -- image-grid-2x2.html
-- views
    -- slider-images.pt.example
```

This theme basically provides you with a copy of the Plone 5 default theme (Barceloneta), and you can change everything you need to create your own theme. The Barceloneta resources are in the folder barceloneta. This is basically a copy of the theme folder of plonetheme.barceloneta. We removed some unneeded files there, because we only need the LESS part for partially including it in our theme.less. We also have the icons and the backend.xml from Barceloneta in our them folder.

In `theme/less` we have our CSS/LESS files. Our own CSS goes into custom.less. You can also add more LESS files and include them in `theme.less`, if you have a lot of custom CSS.

The theme.less is our main LESS file. Here we include all other files we need. It already has some includes of Barceloneta, Bootstrap and our custom.less at the bottom.

We also have a package.json, in which we can define external dependencies like Bootstrap or other CSS/JS packages we want to use in our theme, see *Install external CSS and JavaScript with npm and use them in your theme*.

Start Plone and install your theme product

To start the Plone instance, run:

```
$ ./bin/instance fg
```

The Plone instance will then run on http://localhost:8080. The default username and password is admin / admin. Add a Plone site Plone. Then activate/install your theme product on http://localhost:8080/Plone/prefs_install_products_form. The theme will be automatically enabled. If something is wrong with the theme, you can always go to http://localhost:8080/Plone/@@theming-controlpanel and disable it. This control panel will never be themed, so it works even if the theme might be broken.

Build your Diazo-based theme

You can start with the example files in the theme folder and just change the index.html and custom.less file to customize the default theme to your needs. As stated above it's the Plone 5 default Barceloneta theme plus some custom files you can use to to override or write css/less.

Use your own static mockup

If you got a static mockup from your designer or from a website like http://startbootstrap.com (where the example theme came from), you can use this without customization and just apply the Diazo rules to it.

Another way is to change the static mockup a little bit is to use mostly the same CSS ids and classes. This way it is easier to reuse CSS/LESS from Barceloneta theme and Plone add-ons if needed.

Download and prepare a static theme

Let's start with an untouched static theme, such as this bootstrap theme: http://startbootstrap.com/template-overviews/business-casual/. Just download it and extract it into the theme folder. Replace the index.html with the one in the downloaded theme:

```
$ tree -L 2 .
.
-- about.html
-- backend.xml
-- barceloneta
|   -- less
-- barceloneta-apple-touch-icon-114x114-precomposed.png
-- barceloneta-apple-touch-icon-144x144-precomposed.png
-- barceloneta-apple-touch-icon-57x57-precomposed.png
-- barceloneta-apple-touch-icon-72x72-precomposed.png
-- barceloneta-apple-touch-icon.png
-- barceloneta-apple-touch-icon-precomposed.png
-- barceloneta-favicon.ico
-- blog.html
-- contact.html
-- css
|   -- bootstrap.css
|   -- bootstrap.min.css
|   -- business-casual.css
-- fonts
|   -- glyphicons-halflings-regular.eot
|   -- glyphicons-halflings-regular.svg
|   -- glyphicons-halflings-regular.ttf
|   -- glyphicons-halflings-regular.woff
|   -- glyphicons-halflings-regular.woff2
-- form-handler-nodb.php
-- form-handler.php
-- HOWTO_DEVELOP.rst
-- img
|   -- bg.jpg
|   -- intro-pic.jpg
|   -- slide-1.jpg
|   -- slide-2.jpg
|   -- slide-3.jpg
-- index.html
-- js
|   -- bootstrap.js
|   -- bootstrap.min.js
|   -- jquery.js
-- less
|   -- custom.less
|   -- plone.toolbar.vars.less
|   -- roboto
|   -- theme-compiled.css
|   -- theme-compiled.css.map
|   -- theme.less
|   -- theme.local.less
-- LICENSE
-- manifest.cfg
-- node_modules
|   -- bootstrap
-- package.json
-- preview.png
-- README.md
-- rules.xml
-- template-overrides
-- tinymce-templates
|   -- image-grid-2x2.html
-- views
    -- slider-images.pt.example
```

Preparing the template

To make the given template `index.html` more useful, we customize it a little bit. Right before the second box which contains:

```
<div class="row">
    <div class="box">
        <div class="col-lg-12">
            <hr>
            <h2 class="intro-text text-center">Build a website
```

```
                <strong>worth visiting</strong>
            </h2>
```

Add this:

```
<div class="row">
  <div id="column1-container"></div>
  <div id="content-container">
    <!-- main content (box2 and box3) comes here -->
  </div>
  <div id="column2-container"></div>
</div>
```

And then move the main content (the box 2 and box 3 including the parent row `div`) into the `content-container`.

It should look like this:

```
<div class="row">
  <div id="column1-container"></div>

  <div id="content-container">
      <div class="row">
          <div class="box">
              <div class="col-lg-12">
                  <hr>
                  <h2 class="intro-text text-center">Build a website
                      <strong>worth visiting</strong>
                  </h2>
                  <hr>
                  <img class="img-responsive img-border img-left" src="img/intro-pic.
↪jpg" alt="">
                  <hr class="visible-xs">
                  <p>The boxes used in this template are nested inbetween a normal␣
↪Bootstrap row and the start of your column layout. The boxes will be full-width␣
↪boxes, so if you want to make them smaller then you will need to customize.</p>
                  <p>A huge thanks to <a href="http://join.deathtothestockphoto.com/"␣
↪target="_blank">Death to the Stock Photo</a> for allowing us to use the beautiful␣
↪photos that make this template really come to life. When using this template, make␣
↪sure your photos are decent. Also make sure that the file size on your photos is␣
↪kept to a minumum to keep load times to a minimum.</p>
                  <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nunc␣
↪placerat diam quis nisl vestibulum dignissim. In hac habitasse platea dictumst.␣
↪Interdum et malesuada fames ac ante ipsum primis in faucibus. Pellentesque habitant␣
↪morbi tristique senectus et netus et malesuada fames ac turpis egestas.</p>
              </div>
          </div>
      </div>

      <div class="row">
          <div class="box">
              <div class="col-lg-12">
                  <hr>
                  <h2 class="intro-text text-center">Beautiful boxes
                      <strong>to showcase your content</strong>
                  </h2>
                  <hr>
                  <p>Use as many boxes as you like, and put anything you want in them!
↪ They are great for just about anything, the sky's the limit!</p>
```

```
                  <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nunc␣
↪placerat diam quis nisl vestibulum dignissim. In hac habitasse platea dictumst.␣
↪Interdum et malesuada fames ac ante ipsum primis in faucibus. Pellentesque habitant␣
↪morbi tristique senectus et netus et malesuada fames ac turpis egestas.</p>
              </div>
          </div>
      </div>
  </div>
  <div id="column2-container"></div>
</div>
```

Include theme CSS

We need to include the CSS from the theme into our `theme.less` file:

```
/* theme.less file that will be compiled */

// ### PLONE IMPORTS ###

@barceloneta_path: "barceloneta/less";

//*// Core variables and mixins
@import "@{barceloneta_path}/fonts.plone.less";
@import "@{barceloneta_path}/variables.plone.less";
    @import "@{barceloneta_path}/mixin.prefixes.plone.less";
    @import "@{barceloneta_path}/mixin.tabfocus.plone.less";
    @import "@{barceloneta_path}/mixin.images.plone.less";
    @import "@{barceloneta_path}/mixin.forms.plone.less";
    @import "@{barceloneta_path}/mixin.borderradius.plone.less";
    @import "@{barceloneta_path}/mixin.buttons.plone.less";
    @import "@{barceloneta_path}/mixin.clearfix.plone.less";
//    @import "@{barceloneta_path}/mixin.gridframework.plone.less"; //grid Bootstrap
    @import "@{barceloneta_path}/mixin.grid.plone.less"; //grid Bootstrap

@import "@{barceloneta_path}/normalize.plone.less";
@import "@{barceloneta_path}/print.plone.less";
@import "@{barceloneta_path}/code.plone.less";

//*// Core CSS
@import "@{barceloneta_path}/grid.plone.less";
@import "@{barceloneta_path}/scaffolding.plone.less";
@import "@{barceloneta_path}/type.plone.less";
@import "@{barceloneta_path}/tables.plone.less";
@import "@{barceloneta_path}/forms.plone.less";
@import "@{barceloneta_path}/buttons.plone.less";
@import "@{barceloneta_path}/states.plone.less";

//*// Components
@import "@{barceloneta_path}/breadcrumbs.plone.less";
@import "@{barceloneta_path}/pagination.plone.less";
@import "@{barceloneta_path}/formtabbing.plone.less"; //pattern
@import "@{barceloneta_path}/views.plone.less";
@import "@{barceloneta_path}/thumbs.plone.less";
@import "@{barceloneta_path}/alerts.plone.less";
@import "@{barceloneta_path}/portlets.plone.less";
@import "@{barceloneta_path}/controlpanels.plone.less";
```

```
@import "@{barceloneta_path}/tags.plone.less";
@import "@{barceloneta_path}/contents.plone.less";

//*// Patterns
@import "@{barceloneta_path}/accessibility.plone.less";
@import "@{barceloneta_path}/toc.plone.less";
@import "@{barceloneta_path}/dropzone.plone.less";
@import "@{barceloneta_path}/modal.plone.less";
@import "@{barceloneta_path}/pickadate.plone.less";
@import "@{barceloneta_path}/sortable.plone.less";
@import "@{barceloneta_path}/tablesorter.plone.less";
@import "@{barceloneta_path}/tooltip.plone.less";
@import "@{barceloneta_path}/tree.plone.less";

//*// Structure
@import "@{barceloneta_path}/header.plone.less";
@import "@{barceloneta_path}/sitenav.plone.less";
@import "@{barceloneta_path}/main.plone.less";
@import "@{barceloneta_path}/footer.plone.less";
@import "@{barceloneta_path}/loginform.plone.less";
@import "@{barceloneta_path}/sitemap.plone.less";

//*// Products
@import "@{barceloneta_path}/event.plone.less";
@import "@{barceloneta_path}/image.plone.less";
@import "@{barceloneta_path}/behaviors.plone.less";
@import "@{barceloneta_path}/discussion.plone.less";
@import "@{barceloneta_path}/search.plone.less";

// ### END OF PLONE IMPORTS ###



// ### UTILS ###

// import bootstrap files:
@bootstrap_path: "node_modules/bootstrap/less";

@import "@{bootstrap_path}/variables.less";
@import "@{bootstrap_path}/mixins.less";
@import "@{bootstrap_path}/utilities.less";
@import "@{bootstrap_path}/grid.less";
@import "@{bootstrap_path}/type.less";
@import "@{bootstrap_path}/forms.less";
@import "@{bootstrap_path}/navs.less";
@import "@{bootstrap_path}/navbar.less";
@import "@{bootstrap_path}/carousel.less";

// ### END OF UTILS ###

// include theme css as less
@import (less) "../css/business-casual.css";

// include our custom css/less
@import "custom.less";
```

Here we mainly add the include of the css the theme provides us in `theme/css/business-casual.css` after the END OF UTILS marker, but before the custom.less include. We include the CSS file here as a LESS file. This way we can extend parts of the CSS in our theme, like we will do with the `.box` below.

Note: Don't forget to run `grunt compile` in your package root, after you changed the LESS files or use `grunt watch` to do this automatically after every change!

Using Diazo rules to map the theme with Plone content

Now that we have the static theme, we need to apply the Diazo rules in `rules.xml` to map the Plone content elements to the theme.

First let me explain what we mean when we talk about *content* and *theme*. *Content* is usually the dynamic generated content on the Plone site, and the *theme* is the static template site.

For example:

```
<replace css:theme="#headline" css:content="#firstHeading" />
```

This means that the element `#headline` in the theme should be replaced by the `#firstHeading` element from the generated Plone content.

To inspect the content side, you can open another Browser tab, but instead of http://localhost:8080/Plone, use http://127.0.0.1:8080/Plone. In this tab Diazo is disabled, allowing you to use your browser's Inspector or Developer tools to view the DOM structure of default Plone. This 'unthemed host name' is managed in the Theming control panel > Advanced Settings, where more domains can be added.

For more details on how to use Diazo rules, look at http://docs.diazo.org/en/latest/ and http://docs.plone.org/external/plone.app.theming/docs/index.html.

We already have a fully functional rule set based on the Plone 5 default Theme:

```
<?xml version="1.0" encoding="utf-8"?>
<rules xmlns="http://namespaces.plone.org/diazo"
       xmlns:css="http://namespaces.plone.org/diazo/css"
       xmlns:xhtml="http://www.w3.org/1999/xhtml"
       xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
       xmlns:xi="http://www.w3.org/2001/XInclude">

  <theme href="index.html" />
  <notheme css:if-not-content="#visual-portal-wrapper" />

  <rules css:if-content="#portal-top">
    <!-- Attributes -->
    <copy attributes="*" css:theme="html" css:content="html" />
    <!-- Base tag -->
    <before css:theme="title" css:content="base" />
    <!-- Title -->
    <replace css:theme="title" css:content="title" />
    <!-- Pull in Plone Meta -->
    <after css:theme-children="head" css:content="head meta" />
    <!-- Don't use Plone icons, use the theme's -->
    <drop css:content="head link[rel='apple-touch-icon']" />
    <drop css:content="head link[rel='shortcut icon']" />
    <!-- drop the theme stylesheets -->
    <drop theme="/html/head/link[rel='stylesheet']" />
    <!-- CSS -->
    <after css:theme-children="head" css:content="head link" />
    <!-- Script -->
    <after css:theme-children="head" css:content="head script" />
  </rules>

  <!-- Copy over the id/class attributes on the body tag. This is important for per-
 ↪section styling -->
  <copy attributes="*" css:content="body" css:theme="body" />

  <!-- move global nav -->
  <replace css:theme-children="#mainnavigation" css:content-children="#portal-
 ↪mainnavigation" method="raw" />

  <!-- full-width breadcrumb -->
  <replace css:content="#viewlet-above-content" css:theme="#above-content" />

  <!-- Central column -->
  <replace css:theme="#content-container" method="raw">

    <xsl:variable name="central">
      <xsl:if test="//aside[@id='portal-column-one'] and //aside[@id='portal-column-
 ↪two']">col-xs-12 col-sm-6</xsl:if>
      <xsl:if test="//aside[@id='portal-column-two'] and not(//aside[@id='portal-
 ↪column-one'])">col-xs-12 col-sm-9</xsl:if>
      <xsl:if test="//aside[@id='portal-column-one'] and not(//aside[@id='portal-
 ↪column-two'])">col-xs-12 col-sm-9</xsl:if>
      <xsl:if test="not(//aside[@id='portal-column-one']) and not(//aside[@id='portal-
 ↪column-two'])">col-xs-12 col-sm-12</xsl:if>
    </xsl:variable>

    <div class="{$central}">
      <!-- <p class="pull-right visible-xs">
        <button type="button" class="btn btn-primary btn-xs" data-toggle="offcanvas">
 ↪Toggle nav</button>
      </p> -->
      <div class="row">
        <div class="col-xs-12 col-sm-12">
          <xsl:apply-templates css:select="#content" />
        </div>
      </div>
      <footer class="row">
        <div class="col-xs-12 col-sm-12">
          <xsl:copy-of css:select="#viewlet-below-content" />
        </div>
      </footer>
    </div>
  </replace>

  <!-- Alert message -->
  <replace css:theme-children="#global_statusmessage" css:content-children="#global_
 ↪statusmessage" />

  <!-- Left column -->
  <rules css:if-content="#portal-column-one">
    <replace css:theme="#column1-container">
        <div id="sidebar" class="col-xs-6 col-sm-3 sidebar-offcanvas">
          <aside id="portal-column-one">
              <xsl:copy-of css:select="#portal-column-one > *" />
          </aside>
        </div>
      </replace>
    </rules>

    <!-- Right column -->
    <rules css:if-content="#portal-column-two">
      <replace css:theme="#column2-container">
          <div id="sidebar" class="col-xs-6 col-sm-3 sidebar-offcanvas" role=
↪"complementary">
            <aside id="portal-column-two">
                <xsl:copy-of css:select="#portal-column-two > *" />
            </aside>
          </div>
      </replace>
    </rules>

    <!-- Content header -->
    <replace css:theme="#portal-top" css:content-children="#portal-top" />

    <!-- Footer -->
    <replace css:theme-children="#portal-footer" css:content-children="#portal-footer-
↪wrapper" />

    <!-- toolbar -->
    <replace css:theme="#portal-toolbar" css:content-children="#edit-bar" css:if-not-
↪content=".ajax_load" css:if-content=".userrole-authenticated" />
    <replace css:theme="#anonymous-actions" css:content-children="#portal-personaltools-
↪wrapper" css:if-not-content=".ajax_load" css:if-content=".userrole-anonymous" />

</rules>
```

As you probably noticed, the theme does not look like it should and is missing some important parts like the toolbar. That is because we are using an HTML template, which has different HTML structure, than the one Plone's default theme is using.

We can either change our theme's template to use the same structure and naming for classes and ids, or we can change our rule set to work with the theme template like it is. We will mainly go the second way and customize our rule set to work with the provided theme template. In fact if you use a better theme template then this, where more useful CSS classes and ids used and the grid is defined in CSS/LESS and not in the HTML markup it self, it is a lot easier to work with touching the theme. But we use this popular example theme and therefor need also to make changes to the template it self a bit.

Customize the rule set

The most important part of Plone is the toolbar. So let's first make sure we have it in our theme template.

Plone Toolbar

We already have the needed Diazo rules in our rules.xml:

```
<!-- toolbar -->
<replace css:theme="#portal-toolbar" css:content-children="#edit-bar" css:if-not-
↪content=".ajax_load" css:if-content=".userrole-authenticated" />
```

The only thing we need is a placeholder in our theme template:

```
<section id="portal-toolbar">
</section>
```

You can put it right after the opening body tag in your index.html

Login link & co

If you want to have a login link for your users, you can put this placeholder in your theme template where you want the link to display. You can always log in by adding `/login` to the Plone url, so it's optional.

```
<div id="anonymous-actions">
</div>
```

The necessary rule to fill this with the Plone login link is already in our rules.xml:

```
<replace css:theme="#anonymous-actions" css:content-children="#portal-personaltools-
↪wrapper" css:if-not-content=".ajax_load" css:if-content=".userrole-anonymous" />
```

This will replace your placeholder with `#portal-personaltools-wrapper` from Plone (for example the login link). The link will only be inserted if the user is not already logged in.

Top-navigation

Replace the placeholder with the real Plone top-navigation links. To do this we replace this rule from Barceloneta:

```
<!-- move global nav -->
<replace css:theme-children="#mainnavigation" css:content-children="#portal-
↪mainnavigation" method="raw" />
```

with our new rule:

```
<!-- replace theme navbar-nav with Plone plone-navbar-nav -->
<replace
  css:theme-children=".navbar-nav"
  css:content-children=".plone-navbar-nav" />
```

Here we take the list of links from Plone and replace the placeholder links in the theme with it. The Barceloneta rule copies the whole navigation container into the theme, but only need to copy the links over.

Breadcrumb & co

Plone provides some viewlets like the breadcrumbs (the current path) above the content area.

We already have the needed rule to insert the Plone above-content stuff into the theme:

```
<!-- full-width breadcrumb -->
<replace css:content="#viewlet-above-content" css:theme="#above-content" />
```

To get this into the theme layout, we add a placeholder with the CSS id `#above-content` to the theme's `index.html`. This is the place where we want to insert Plone's "above-content" stuff.

For example, at the top of the `div.container` after:

```
<!-- Navigation -->
<nav class="navbar navbar-default" role="navigation">
    ...
</nav>

<div class="container">

    <!-- insert here -->
```

goes this before the row/box:

```
<div class="row">
    <div id="above-content" class="box"></div>
</div>
```

This will bring over everything from the `viewlet-above-content` block from Plone.

This also includes the Breadcrumb bar. Because our current theme does not provide a breadcrumb bar, we can just drop it from the Plone content, like this:

```
<drop css:content="#portal-breadcrumbs" />
```

If you only want to drop this for non-administrators, you can do it like this:

```
<drop
 css:content="#portal-breadcrumbs"
 css:if-not-content=".userrole-manager"
 />
```

Or for anonymous users only:

```
<drop
 css:content="#portal-breadcrumbs"
 css:if-content=".userrole-anonymous"
 />
```

Note: The classes like *userrole-anonymous* are provided by Plone in the `body` tag.

Slider only on Front-page

We want the slider in the template only on the front page, and we don't want it when we are editing the front page. To make this easier, we add `#front-page-slider` to the outer row `div`-tag which contains the slider:

```
<div class="row" id="front-page-slider">
    <div class="box">
        <div class="col-lg-12 text-center">
            <div id="carousel-example-generic" class="carousel slide">
                <!-- Indicators -->
                <ol class="carousel-indicators hidden-xs">
                    <li data-target="#carousel-example-generic" data-slide-to="0"␣
↪class="active"></li>
                    <li data-target="#carousel-example-generic" data-slide-to="1"></
↪li>
                    <li data-target="#carousel-example-generic" data-slide-to="2"></
↪li>
```

```
                </ol>

                <!-- Wrapper for slides -->
                <div class="carousel-inner">
                    <div class="item active">
                        <img class="img-responsive img-full" src="img/slide-1.jpg"␣
↪alt="">
                    </div>
                    <div class="item">
                        <img class="img-responsive img-full" src="img/slide-2.jpg"␣
↪alt="">
                    </div>
                    <div class="item">
                        <img class="img-responsive img-full" src="img/slide-3.jpg"␣
↪alt="">
                    </div>
                </div>

                <!-- Controls -->
                <a class="left carousel-control" href="#carousel-example-generic"␣
↪data-slide="prev">
                    <span class="icon-prev"></span>
                </a>
                <a class="right carousel-control" href="#carousel-example-generic"␣
↪data-slide="next">
                    <span class="icon-next"></span>
                </a>
            </div>
            <h2 class="brand-before">
                    <small>Welcome to</small>
            </h2>
            <h1 class="brand-name">Business Casual</h1>
            <hr class="tagline-divider">
            <h2>
                <small>By
                    <strong>Start Bootstrap</strong>
                </small>
            </h2>
          </div>
    </div>
</div>
```

Now we can drop it if we are not on the front page and also in some other situations:

```
<drop
  css:theme="#front-page-slider"
  css:if-not-content=".section-front-page.template-document_view" />
```

Currently the slider is still static, but we will change that later in *Create dynamic slider content in Plone*.

Title and Description

Let's delete the tag with the id `brand-before` from the theme template.

```
<drop
  css:theme=".brand-before"
  css:if-content=".section-front-page" />
```

Now let's put the necessary rules for the Title and Description in our rules.xml:

```
<replace
  css:theme-children=".brand-name"
  css:content-children=".documentFirstHeading"
  method="raw" />
<drop
  css:content=".documentFirstHeading"
  css:if-content=".section-front-page" />

<replace
  css:theme="#front-page-slider h2"
  css:content=".documentDescription"
  method="raw" />
<drop
  css:content=".documentDescription"
  css:if-content=".section-front-page" />
```

If we have the slider on the front page, the Plone title will be placed inside the tag with the class `brand-name`. If we don't have the slider, we see the title inside the tag with the class `documentFirstHeading`.

Status messages

Plone will render status messages in the `#global_statusmessage` element. We want to bring these messages across to the theme. For this, we add another placeholder into our theme template:

```
<div class="row">
    <div id="global_statusmessage"></div>
    <div id="above-content"></div>
</div>
```

and we already have this rule to bring the messages across:

```
<!-- Alert message -->
<replace css:theme-children="#global_statusmessage" css:content-children="#global_
↪statusmessage" />
```

To test that, just edit the front page. You should see a confirmation message from Plone.

Main content area 1

To make the Plone content area flexible and containing the correct bootstrap grid classes, we use an inline XSL snippet. This is already in our rules.xml, but needs some customization for our theme:

```
<!-- Central column -->
<replace css:theme="#content-container" method="raw">

  <xsl:variable name="central">
    <xsl:if test="//aside[@id='portal-column-one'] and //aside[@id='portal-column-two
↪']">col-xs-12 col-sm-6</xsl:if>
    <xsl:if test="//aside[@id='portal-column-two'] and not(//aside[@id='portal-column-
↪one'])">col-xs-12 col-sm-9</xsl:if>
    <xsl:if test="//aside[@id='portal-column-one'] and not(//aside[@id='portal-column-
↪two'])">col-xs-12 col-sm-9</xsl:if>
```

```
    <xsl:if test="not(//aside[@id='portal-column-one']) and not(//aside[@id='portal-
↪column-two'])">col-xs-12 col-sm-12</xsl:if>
  </xsl:variable>

  <div class="{$central}">
    <!-- <p class="pull-right visible-xs">
      <button type="button" class="btn btn-primary btn-xs" data-toggle="offcanvas">
↪Toggle nav</button>
    </p> -->
    <div class="row">
      <div class="box">
        <div class="col-xs-12 col-sm-12">
          <xsl:apply-templates css:select="#content" />
        </div>
      </div>
    </div>
    <footer class="row">
      <div class="box">
        <div class="col-xs-12 col-sm-12">
          <xsl:copy-of css:select="#viewlet-below-content" />
        </div>
      </div>
    </footer>
  </div>
</replace>
```

This will add the right grid classes to the content columns depending on one-column-, two-column- or three-column-layout. We need to wrap these elements in a div with the class `box`.

Left and right columns

We have already added the `column1-container` and `column2-container` ids to our template. The following rules will incorporate the left and the right columns from Plone into the theme, and also change their markup to be an `aside` instead of a normal `div`. That is the reason to use inline XSL here, but we already have it in our rules:

```
<!-- Left column -->
<rules css:if-content="#portal-column-one">
  <replace css:theme="#column1-container">
      <div id="left-sidebar" class="col-xs-6 col-sm-3 sidebar-offcanvas">
        <aside id="portal-column-one">
           <xsl:copy-of css:select="#portal-column-one > *" />
        </aside>
      </div>
  </replace>
</rules>

<!-- Right column -->
<rules css:if-content="#portal-column-two">
  <replace css:theme="#column2-container">
      <div id="right-sidebar" class="col-xs-6 col-sm-3 sidebar-offcanvas" role=
↪"complementary">
        <aside id="portal-column-two">
           <xsl:copy-of css:select="#portal-column-two > *" />
        </aside>
      </div>
  </replace>
```

```
</rules>
```

So nothing more to do here.

Footer

Bring across the footer from Plone:

```
<!-- footer -->
<replace
  css:theme-children="footer > .container"
  css:content-children="#portal-footer-wrapper" />
```

That was basically all to bring the theme together with the dynamic elements from Plone. The rest is more or less CSS. Later we will *Create dynamic slider content in Plone* to make the slider dynamic and let users change the pictures for the slider.

Understanding and using the Grunt build system

We already have a `Gruntfile.js` in the top level directory of our theme package:

```
module.exports = function (grunt) {
    'use strict';
    grunt.initConfig({
        pkg: grunt.file.readJSON('package.json'),
        // we could just concatenate everything, really
        // but we like to have it the complex way.
        // also, in this way we do not have to worry
        // about putting files in the correct order
        // (the dependency tree is walked by r.js)
        less: {
            dist: {
                options: {
                    paths: [],
                    strictMath: false,
                    sourceMap: true,
                    outputSourceFiles: true,
                    sourceMapURL: '++theme++tango/less/theme-compiled.css.map',
                    sourceMapFilename: 'less/theme-compiled.css.map',
                    modifyVars: {
                        "isPlone": "false"
                    }
                },
                files: {
                    'less/theme-compiled.css': 'less/theme.local.less',
                }
            }
        },
        postcss: {
            options: {
                map: true,
                processors: [
                    require('autoprefixer')({
                        browsers: ['last 2 versions']
                    })
                ]
            },
            dist: {
                src: 'less/*.css'
            }
        },
        watch: {
            scripts: {
                files: [
                    'less/*.less',
                    'barceloneta/less/*.less'
                ],
                tasks: ['less', 'postcss']
            }
        },
        browserSync: {
            html: {
                bsFiles: {
                    src : [
                      'less/*.less',
                      'barceloneta/less/*.less'
                    ]
                },
                options: {
                    watchTask: true,
                    debugInfo: true,
                    online: true,
                    server: {
                        baseDir: "."
                    },
                }
            },
            plone: {
                bsFiles: {
                    src : [
                      'less/*.less',
                      'barceloneta/less/*.less'
                    ]
                },
                options: {
                    watchTask: true,
                    debugInfo: true,
                    proxy: "localhost:8080",
                    reloadDelay: 3000,
                    // reloadDebounce: 2000,
                    online: true
                }
            }
        }
    });

    // grunt.loadTasks('tasks');
    grunt.loadNpmTasks('grunt-browser-sync');
    grunt.loadNpmTasks('grunt-contrib-watch');
    grunt.loadNpmTasks('grunt-contrib-less');
    grunt.loadNpmTasks('grunt-postcss');

    // CWD to theme folder
    grunt.file.setBase('./src/plonetheme/tango/theme');

    grunt.registerTask('compile', ['less', 'postcss']);
    grunt.registerTask('default', ['compile']);
    grunt.registerTask('bsync', ["browserSync:html", "watch"]);
    grunt.registerTask('plone-bsync', ["browserSync:plone", "watch"]);
};
```

At the end, we can see some registered `Grunt` tasks. We can use these tasks to control what happens when we run `Grunt`.

By default `Grunt` will just run the `compile task`, which means the less files are getting compiled and the postcss task is run:

```
$ grunt
Running "less:dist" (less) task
>> 1 stylesheet created.
>> 1 sourcemap created.

Running "postcss:dist" (postcss) task
>> 1 processed stylesheet created.

Done, without errors.
```

If we want `grunt` to watch for changes in our less files and let them compile it automatically after every change, we can run `grunt watch`, and it will run the `compile` task after every change of a LESS file:

```
$ grunt watch
Running "watch" task
Waiting...
```

If some LESS file has changed, you will see something like this:

```
$ grunt watch
Running "watch" task
Waiting...
>> File "less/custom.less" changed.
Running "less:dist" (less) task
>> 1 stylesheet created.
>> 1 sourcemap created.

Running "postcss:dist" (postcss) task
>> 1 processed stylesheet created.

Done, without errors.
Completed in 2.300s at Mon Oct 10 2016 20:05:27 GMT+0200 (CEST) - Waiting...

Done, without errors.
```

They are also other useful tasks like `plone-bsync`, which we can use to also update the Browser after changes.

```
$ grunt plone-bsync
Running "browserSync:plone" (browserSync) task
[BS] Proxying: http://localhost:8081
[BS] Access URLs:
 --------------------------------------
       Local: http://localhost:3000
    External: http://192.168.2.149:3000
 --------------------------------------
          UI: http://localhost:3001
 UI External: http://192.168.2.149:3001
 --------------------------------------
[BS] Watching files...

Running "watch" task
Waiting...
```

You will now see an open browser window, which is automatically reloaded any time a LESS file has changed and the CSS was recompiled.

Note: If you use other ports or IP's for your Plone backend, you have to set up the proxy in the Gruntfile.js to the Plone backend address:port.

Theme manifest.xml

Now let's have a look at our theme's `manifest.cfg` which declares `development-css`, `production-css` and optionally `tinymce-content-css`, like this:

```
[theme]
title = Plone Theme: Tango
description = A Diazo based Plone theme
doctype = <!DOCTYPE html>
rules = /++theme++tango/rules.xml
prefix = /++theme++tango
enabled-bundles =
disabled-bundles =

development-css = /++theme++tango/less/theme.less
production-css = /++theme++tango/less/theme-compiled.css
tinymce-content-css = /++theme++tango/less/theme-compiled.css

# development-js = /++theme++tango/js/theme.js
# production-js = /++theme++tango/js/theme-compiled.js

[theme:overrides]
directory = template-overrides

[theme:parameters]
# portal_url = python: portal.absolute_url()
```

The `development-css` file is used when Plone is running in development mode, otherwise the file under `production-css` will be used.

The last one `tinymce-content-css` tells Plone to load that particular CSS file inside TinyMCE, wherever a TinyMCE rich text field is displayed.

Note: After making manifest changes, we need to deactivate/activate the theme for them to take effect. Just go to `/@@theming-controlpanel` and do it.

Final CSS customization

To make our theme look nicer, we add some CSS as follows to our `custom.less` file:

```
/* Custom LESS file that is included from the theme.less file */

.brand-name{
    margin-top: 0.5em;
}

.documentDescription{
    margin-top: 1em;
}

.clearFix{
    clear: both;
}

#left-sidebar {
    padding-left: 0;
}

#right-sidebar {
    padding-right: 0;
}

#content {
    label, .label {
        color: #333;
        font-size: 100%;
    }
}

.pat-autotoc.autotabs, .autotabs {
    border-width: 0;
}

.portal-column-one .portlet,
.portal-column-two .portlet {
    .box;
}

footer .portletActions{
}

footer {
    .portlet {
        padding: 1em 0;
        margin-bottom: 0;
        border: 0;
        background: transparent;
        .portletContent{
            border: 0;
            background: transparent;
            ul {
                padding-left: 0;
                list-style-type: none;
                .portletItem {
                    display: inline-block;
                    &:not(:last-child){
                        padding-right: 0.5em;
                        margin-right: 0.5em;
                        border-right: 1px solid;
                    }
                    &:hover{
                        background-color: transparent;
                    }
                    a{
                        color: #000;
                        padding: 0;
                        text-decoration: none;
                        &:hover{
                            background-color: transparent;
                        }
                        &::before{
                            content: none;
                        }
                    }
                }
            }
        }
    }
}
```

Install external CSS and JavaScript with npm and use them in your theme

As our theme is based on `Bootstrap`, we want to install `Bootstrap` with `npm` to have more flexibility, for example to use the LESS file of Bootstrap. To do that, we use `npm`, which you should already have globally installed on your system.

Note: The following steps are already included in bobtemplates.plone template, they are here only for documentation reasons, to show how to install and use external packages like `Bootstrap`.

To install `Bootstrap` with `npm` run the following command inside the theme folder:

```
$ npm install bootstrap --save
```

The `--save` option will add the package to `package.json` in the theme folder for us. Now, we can install all dependencies on any other system by running the following command from inside of our theme folder:

```
$ npm install
```

Now that we have installed bootstrap using npm, we have all bootstrap components available in the subfolder called `node_modules`:

```
$ tree node_modules/bootstrap/
node_modules/bootstrap/
-- CHANGELOG.md
-- dist
|   -- css
|   |   -- bootstrap.css
|   |   -- bootstrap.css.map
|   |   -- bootstrap.min.css
|   |   -- bootstrap-theme.css
|   |   -- bootstrap-theme.css.map
|   |   -- bootstrap-theme.min.css
|   -- fonts
|   |   -- glyphicons-halflings-regular.eot
|   |   -- glyphicons-halflings-regular.svg
|   |   -- glyphicons-halflings-regular.ttf
|   |   -- glyphicons-halflings-regular.woff
|   |   -- glyphicons-halflings-regular.woff2
|   -- js
|       -- bootstrap.js
|       -- bootstrap.min.js
|       -- npm.js
-- fonts
|   -- glyphicons-halflings-regular.eot
|   -- glyphicons-halflings-regular.svg
|   -- glyphicons-halflings-regular.ttf
|   -- glyphicons-halflings-regular.woff
|   -- glyphicons-halflings-regular.woff2
-- grunt
|   -- bs-commonjs-generator.js
|   -- bs-glyphicons-data-generator.js
|   -- bs-lessdoc-parser.js
|   -- bs-raw-files-generator.js
|   -- configBridge.json
|   -- sauce_browsers.yml
-- Gruntfile.js
-- js
|   -- affix.js
|   -- alert.js
|   -- button.js
|   -- carousel.js
|   -- collapse.js
|   -- dropdown.js
|   -- modal.js
|   -- popover.js
|   -- scrollspy.js
|   -- tab.js
|   -- tooltip.js
|   -- transition.js
-- less
|   -- alerts.less
|   -- badges.less
|   -- bootstrap.less
|   -- breadcrumbs.less
|   -- button-groups.less
|   -- buttons.less
|   -- carousel.less
|   -- close.less
|   -- code.less
|   -- component-animations.less
|   -- dropdowns.less
|   -- forms.less
|   -- glyphicons.less
|   -- grid.less
|   -- input-groups.less
|   -- jumbotron.less
|   -- labels.less
|   -- list-group.less
|   -- media.less
|   -- mixins
|   |   -- alerts.less
|   |   -- background-variant.less
|   |   -- border-radius.less
|   |   -- buttons.less
|   |   -- center-block.less
|   |   -- clearfix.less
|   |   -- forms.less
|   |   -- gradients.less
|   |   -- grid-framework.less
|   |   -- grid.less
|   |   -- hide-text.less
|   |   -- image.less
|   |   -- labels.less
|   |   -- list-group.less
|   |   -- nav-divider.less
|   |   -- nav-vertical-align.less
|   |   -- opacity.less
|   |   -- pagination.less
|   |   -- panels.less
|   |   -- progress-bar.less
|   |   -- reset-filter.less
|   |   -- reset-text.less
|   |   -- resize.less
|   |   -- responsive-visibility.less
|   |   -- size.less
|   |   -- tab-focus.less
|   |   -- table-row.less
|   |   -- text-emphasis.less
|   |   -- text-overflow.less
|   |   -- vendor-prefixes.less
|   -- mixins.less
|   -- modals.less
|   -- navbar.less
|   -- navs.less
|   -- normalize.less
|   -- pager.less
|   -- pagination.less
|   -- panels.less
|   -- popovers.less
|   -- print.less
|   -- progress-bars.less
|   -- responsive-embed.less
|   -- responsive-utilities.less
|   -- scaffolding.less
|   -- tables.less
|   -- theme.less
|   -- thumbnails.less
|   -- tooltip.less
|   -- type.less
|   -- utilities.less
|   -- variables.less
|   -- wells.less
-- LICENSE
-- package.json
-- README.md
```

To include the needed “carousel” part and some other bootstrap components which our downloaded theme uses, we change our `theme.less` to look like this:

```
/* theme.less file that will be compiled */

/* ### PLONE IMPORTS ### */

@barceloneta_path: "barceloneta/less";

/* Core variables and mixins */
@import "@{barceloneta_path}/fonts.plone.less";
@import "@{barceloneta_path}/variables.plone.less";
    @import "@{barceloneta_path}/mixin.prefixes.plone.less";
    @import "@{barceloneta_path}/mixin.tabfocus.plone.less";
    @import "@{barceloneta_path}/mixin.images.plone.less";
    @import "@{barceloneta_path}/mixin.forms.plone.less";
    @import "@{barceloneta_path}/mixin.borderradius.plone.less";
    @import "@{barceloneta_path}/mixin.buttons.plone.less";
    @import "@{barceloneta_path}/mixin.clearfix.plone.less";
//    @import "@{barceloneta_path}/mixin.gridframework.plone.less"; //grid Bootstrap
    @import "@{barceloneta_path}/mixin.grid.plone.less"; //grid Bootstrap

@import "@{barceloneta_path}/normalize.plone.less";
@import "@{barceloneta_path}/print.plone.less";
@import "@{barceloneta_path}/code.plone.less";

/* Core CSS */
@import "@{barceloneta_path}/grid.plone.less";
@import "@{barceloneta_path}/scaffolding.plone.less";
@import "@{barceloneta_path}/type.plone.less";
@import "@{barceloneta_path}/tables.plone.less";
@import "@{barceloneta_path}/forms.plone.less";
@import "@{barceloneta_path}/buttons.plone.less";
@import "@{barceloneta_path}/states.plone.less";

/* Components */
@import "@{barceloneta_path}/breadcrumbs.plone.less";
@import "@{barceloneta_path}/pagination.plone.less";
@import "@{barceloneta_path}/formtabbing.plone.less"; //pattern
@import "@{barceloneta_path}/views.plone.less";
@import "@{barceloneta_path}/thumbs.plone.less";
@import "@{barceloneta_path}/alerts.plone.less";
@import "@{barceloneta_path}/portlets.plone.less";
@import "@{barceloneta_path}/controlpanels.plone.less";
@import "@{barceloneta_path}/tags.plone.less";
@import "@{barceloneta_path}/contents.plone.less";

/* Patterns */
@import "@{barceloneta_path}/accessibility.plone.less";
@import "@{barceloneta_path}/toc.plone.less";
@import "@{barceloneta_path}/dropzone.plone.less";
@import "@{barceloneta_path}/modal.plone.less";
@import "@{barceloneta_path}/pickadate.plone.less";
@import "@{barceloneta_path}/sortable.plone.less";
@import "@{barceloneta_path}/tablesorter.plone.less";
@import "@{barceloneta_path}/tooltip.plone.less";
```

```
@import "@{barceloneta_path}/tree.plone.less";

/* Structure */
@import "@{barceloneta_path}/header.plone.less";
@import "@{barceloneta_path}/sitenav.plone.less";
@import "@{barceloneta_path}/main.plone.less";
@import "@{barceloneta_path}/footer.plone.less";
@import "@{barceloneta_path}/loginform.plone.less";
@import "@{barceloneta_path}/sitemap.plone.less";

/* Products */
@import "@{barceloneta_path}/event.plone.less";
@import "@{barceloneta_path}/image.plone.less";
@import "@{barceloneta_path}/behaviors.plone.less";
@import "@{barceloneta_path}/discussion.plone.less";
@import "@{barceloneta_path}/search.plone.less";

// ### END OF PLONE IMPORTS ###

// ### UTILS ###

// import bootstrap files:
@bootstrap_path: "node_modules/bootstrap/less";

@import "@{bootstrap_path}/variables.less";
@import "@{bootstrap_path}/mixins.less";
@import "@{bootstrap_path}/utilities.less";
@import "@{bootstrap_path}/grid.less";
@import "@{bootstrap_path}/type.less";
@import "@{bootstrap_path}/forms.less";
@import "@{bootstrap_path}/navs.less";
@import "@{bootstrap_path}/navbar.less";
@import "@{bootstrap_path}/carousel.less";

// END OF UTILS

// include theme css as less
@import (less) "../css/business-casual.css";

// include our custom css/less
@import "custom.less";
```

Here you can see how we include the resources like `@import "@{bootstrap_path}/carousel.less";` in our LESS file.

Also take notice of the definition:

```
@bootstrap_path: "node_modules/bootstrap/less";
```

here we define the bootstrap path, so that we can use it in all bootstrap includes.

Note: Don't forget to run `grunt compile` after you changed the LESS files or use `grunt watch` to do this automatically after every change!

More Diazo and plone.app.theming details

For more details how to build a Diazo based theme, look at http://docs.diazo.org/en/latest/ and http://docs.plone.org/external/plone.app.theming/docs/index.html.

Creating and customizing Plone templates

Overriding a Plone template

A large part of the Plone UI is provided by BrowserView and Viewlet templates.

You can see all viewlets and their managers (sortable containers) when you view the URL `./@@manage-viewlets`).

Note: To override them from the ZMI, you can go to `./portal_view_customizations`. But this is very limited and does not work for all views.

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 `bobtemplates.plone` theme skeleton via `plone.app.themingplugins`, you can start using it immediately by adding all the templates you want to override in the `src/plonetheme/tango/theme/template-overrides` directory.

In order for jbot to match the override to the template which is being overridden, the name of the *new* template needs to include the complete path to the original template as a prefix (with every `/` replaced by `.`).

For instance, to override `path_bar.pt` (the breadcrumbs) from `plone.app.layout`, knowing that this template is found in a sub folder named `viewlets`, you need to name the overriding template `plone.app.layout.viewlets.path_bar.pt`.

Note: Clicking the template in ZMI > portal_view_customizations is a handy way to find the template path. You can also copy the original template's code here.

When a new override has been added, the Plone instance needs to be restarted. After this, you can just refresh the page to see any changes to the template.

Overriding Event Item template

The path to the original template is `plone/app/event/browser/event_view.pt`, so the full dotted name for our replacement template should be: `plone.app.event.browser.event_view.pt`. Create a new file with this dotted name into the `template-overrides` folder.

Let's say we want to move the full text of the event item to appear before the event details block. To do this, we copy over the original template code and change the order of the two blocks:

Note: If your buildout is using `omelette`, you can find the original template in `buildout/parts/omelette/plone/app/event/browser`.

```
<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="context/main_template/macros/master"
      i18n:domain="plone.app.event">
<body>

<metal:content-core fill-slot="content-core">
<metal:block define-macro="content-core">
<tal:def tal:define="data nocall:view/data">

  <div class="event clearfix" itemscope="itemscope" itemtype="http://data-vocabulary.
↪org/Event">

    <ul class="hiddenStructure">
      <li><a itemprop="url" class="url" href="" tal:attributes="href data/url" tal:
↪content="data/url">url</a></li>
      <li itemprop="summary" class="summary" tal:content="data/title">title</li>
      <li itemprop="startDate" class="dtstart" tal:content="data/start/isoformat">
↪start</li>
      <li itemprop="endDate" class="dtend" tal:content="data/end/isoformat">end</li>
      <li itemprop="description" class="description" tal:content="data/description">
↪description</li>
    </ul>

    <div id="parent-fieldname-text" tal:condition="data/text">
      <tal:text content="structure data/text" />
    </div>

    <tal:eventsummary replace="structure context/@@event_summary"/>

  </div>

</tal:def>
</metal:block>
</metal:content-core>

</body>
</html>
```

You can now restart Plone and view an event to see the effect.

Creating a new Plone template

Create dynamic slider content in Plone

To render our dynamic content for the slider we need a custom view in Plone. There are various ways to create Views. For now, we will use a very simple template-only-view via jbot and `themingplugins`. The `bobtemplates.plone` skeleton includes everything you need.

The only thing we need to do, is to add a template file in the `theme/views` folder. Here we create a template file named `slider-images.pt`. Luckily we already have this file as an example. So the only thing we need to do is, to rename the file `slider-images.pt.examle```to ``slider-images.pt`.

```
$ tree views/
views/
-- slider-images.pt.examle
```

The template code looks like this:

```
<div id="carousel-example-generic" class="carousel slide">
 <!-- Indicators -->
 <ol class="carousel-indicators hidden-xs">
     <li tal:repeat="item context/keys"
         data-target="#carousel-example-generic"
         data-slide-to="${python:repeat.item.index}"
         class="${python: repeat.item.start and 'active' or ''}"></li>
 </ol>

 <!-- Wrapper for slides -->
 <div class="carousel-inner">
     <div tal:repeat="item context/values"
         class="item ${python: repeat.item.start and 'active' or ''}">
         <img tal:define="scales item/@@images"
             tal:replace="structure python: scales.tag('image', scale='large', css_class='img-responsive img-full')" />
     </div>
 </div>

 <!-- Controls -->
 <a class="left carousel-control" href="#carousel-example-generic" data-slide="prev">
     <span class="icon-prev"></span>
 </a>
 <a class="right carousel-control" href="#carousel-example-generic" data-slide="next">
     <span class="icon-next"></span>
 </a>
</div>
```

This is all that's required to create a very simple template-only View. You can test the view after restarting your Plone instance. For the View to show up, it needs some images to display. To supply the images, we have to create a folder in Plone named `slider-images` and add some images there.

Note: we will show you later how to *Creating initial content for the theme*

Now we can browse to the View on this folder by visiting: http://localhost:8080/Plone/slider-images/@@slider-images This will render the markup required to render the slider.

Take over the dynamic slider content from Plone

Now that we have our `slider-images` View which renders our HTML markup for the slider, we need to include that on the front page. For that, we use Diazo's ability to load the content from other URLs, using the `href` attribute in our `rules.xml`:

```
<!-- dynamic slider content -->
<replace
  css:theme="#carousel-example-generic"
  css:content="#carousel-example-generic"
  href="/slider-images/@@slider-images" />
```

Creating initial content for the theme

Our theme relies on some initial content structure, specifically the `slider-images` folder with some images inside. Let's improve our theme package to create this content on install.

To do that we create the `slider-images` folder in our `setuphandlers.py` and load also some example images into that folder.

We have the needed images inside `theme/img` folder. To create the folder and the immages put the following code in your setuphandlers.py.

```
# -*- coding: utf-8 -*-

from plone import api
from Products.CMFPlone.interfaces import INonInstallable
from zope.interface import implementer
import os


@implementer(INonInstallable)
class HiddenProfiles(object):

    def getNonInstallableProfiles(self):
        """Hide uninstall profile from site-creation and quickinstaller"""
        return [
            'plonetheme.tango:uninstall',
        ]


def post_install(context):
    """Post install script"""
    portal = api.portal.get()
    _create_content(portal)


def _create_content(portal):
    if not portal.get('slider-images', False):
        slider = api.content.create(
            type='Folder',
            container=portal,
            title=u'Slider',
            id='slider-images'
        )
        for slider_number in range(1, 4):
            slider_name = u'slider-{0}'.format(str(slider_number))
            slider_image = api.content.create(
                type='Image',
                container=slider,
                title=slider_name,
                id=slider_name
            )
            slider_image.image = _load_image(slider_number)
        # NOTE: if your plone site is not a vanilla plone
        # you can have different workflows on folders and images
        # or different transitions names so this could fail
        # and you'll need to publish the images as well
        # or do that manually TTW.
        api.content.transition(obj=slider, transition='publish')
```

```
def _load_image(slider):
    from plone.namedfile.file import NamedBlobImage
    filename = os.path.join(os.path.dirname(__file__), 'theme', 'img',
                            'slide-{0}.jpg'.format(slider))
    return NamedBlobImage(
        data=open(filename, 'r').read(),
        filename=u'slide-{0}.jpg'.format(slider)
    )


def uninstall(context):
    """Uninstall script"""
```

Note: After adding this code to the setuphandlers.py, we need to restart Plone and uninstall/install our theme package.

Using TinyMCE templates

TinyMCE has a *templates* plugin which provides an easy way to create complex content in TinyMCE. You can use that to help users to add complex content structures like predefined tables or content. The users then only need to customize this content to their needs.

Create your own TinyMCE templates

We already have a folder named `tinymce-templates` in our theme folder. So we just need to add a file named `content-box.html` into the this folder:

```
maik@planetmobile:~/develop/plone/plonetheme.tango/src/plonetheme/tango/theme
$ tree tinymce-templates/
tinymce-templates/
-- content-box.html
```

In the file `content-box.html` we put this HTML template content:

```
<div class="mceTmpl">
    <div class="row">
        <div class="box">
            <div class="col-lg-12">
                <hr>
                <h2 class="intro-text text-center">Build a website
                    <strong>worth visiting</strong>
                </h2>
                <hr>
                <hr class="visible-xs">
                <p>The boxes used in this template are nested between a normal␣
↪Bootstrap row and the start of your column layout. The boxes will be full-width␣
↪boxes, so if you want to make them smaller then you will need to customize.</p>
                <p>A huge thanks to <a href="http://join.deathtothestockphoto.com/"␣
↪target="_blank">Death to the Stock Photo</a> for allowing us to use the beautiful␣
↪photos that make this template really come to life. When using this template, make␣
↪sure your photos are decent. Also make sure that the file size on your photos is␣
↪kept to a minumum to keep load times to a minimum.</p>
                <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nunc␣
↪placerat diam quis nisl vestibulum dignissim. In hac habitasse platea dictumst.␣
↪Interdum et malesuada fames ac ante ipsum primis in faucibus. Pellentesque habitant␣
↪morbi tristique senectus et netus et malesuada fames ac turpis egestas.</p>
```

```
      </div>
    </div>
  </div>
</div>
```

This is the template content we will get in TinyMCE when we use this template.

Activate TinyMCE templates plugin

Note: The activation of the TinyMCE template plugin is already provided by bobtemplates.plone, the only thing you have to do is to add your template tothe registry, like described below.

If the -plugin is not already activated, you can activate the template plugin (the `custom_plugins` record) and register this template for TinyMCE (the `template` record):

```
<?xml version="1.0"?>
<registry>
  <!-- register our template -->
  <record name="plone.templates" interface="Products.CMFPlone.interfaces.controlpanel.
↪ITinyMCESchema" field="templates">
    <field type="plone.registry.field.Text">
      <default></default>
      <description xmlns:ns0="http://xml.zope.org/namespaces/i18n" ns0:domain="plone"␣
↪ns0:translate="help_tinymce_templates">Enter the list of templates in json format ␣
↪              http://www.tinymce.com/wiki.php/Plugin:template</description>
      <required>False</required>
      <title xmlns:ns0="http://xml.zope.org/namespaces/i18n" ns0:domain="plone" ns0:
↪translate="label_tinymce_templates">Templates</title>
    </field>
    <value>[
      {"title": "Image Grid 2x2", "url": "++theme++tango/tinymce-templates/image-grid-
↪2x2.html"},
      {"title": "Content box", "url": "++theme++tango/tinymce-templates/content-box.
↪html"}
      ]
    </value>
  </record>

  <!-- activate the plugin -->
  <record name="plone.custom_plugins" interface="Products.CMFPlone.interfaces.
↪controlpanel.ITinyMCESchema" field="custom_plugins">
      <field type="plone.registry.field.List">
        <default/>
        <description xmlns:ns0="http://xml.zope.org/namespaces/i18n" ns0:domain="plone
↪" ns0:translate="">Enter a list of custom plugins which will be loaded in the␣
↪editor. Format is pluginname|location, one per line.</description>
        <required>False</required>
        <title xmlns:ns0="http://xml.zope.org/namespaces/i18n" ns0:domain="plone" ns0:
↪translate="">Custom plugins</title>
        <value_type type="plone.registry.field.TextLine"/>
      </field>
      <value>
        <element>template|+plone+static/components/tinymce-builded/js/tinymce/plugins/
↪template</element>
      </value>
  </record>
</registry>
```

As we already have the configuration already in place and even a TinyMCE template already exists, we only need to extend the following list with our template file.

```
<value>[
 {"title": "Image Grid 2x2", "url": "++theme++tango/tinymce-templates/image-grid-2x2.
↪html"},
 {"title": "Content box", "url": "++theme++plonetheme.tango/tinymce-templates/content-
↪box.html"}
 ]
</value>
```

Note: Now you need to uninstall/install (or import the registry profile from `portal_setup`) your package to update the registry configuration.

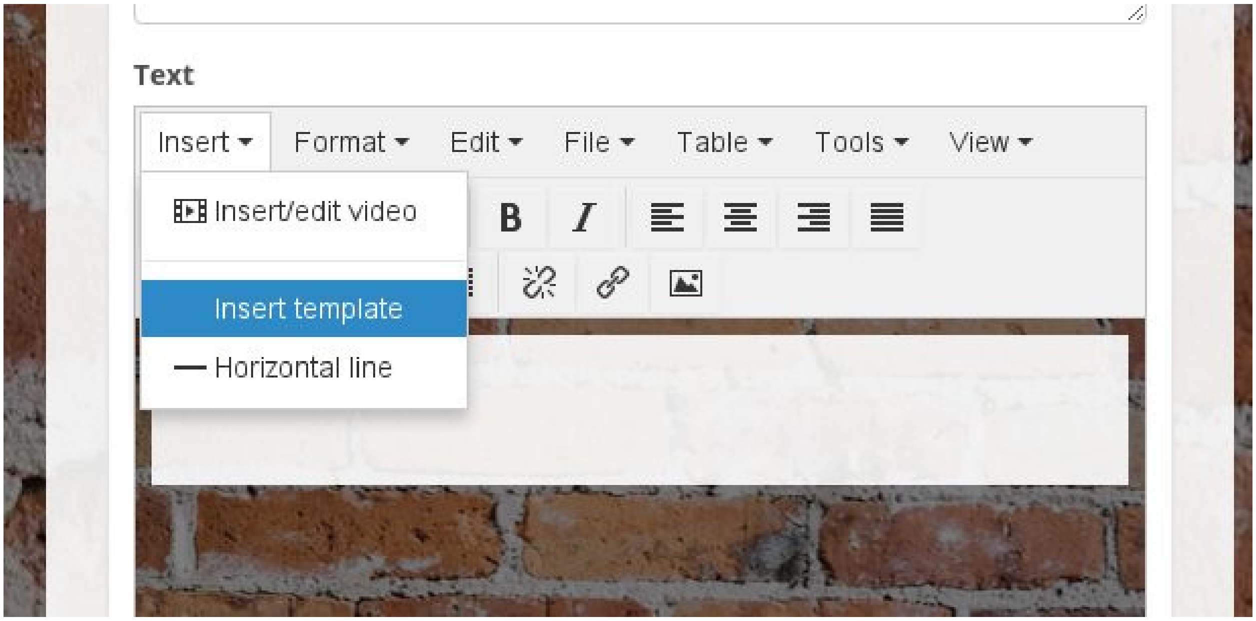
You can also add the template TTW in the TinyMCE control panel by updating the following snippet:

```
[
  {
   "title": "Image Grid 2x2",
   "url": "++theme++tango/tinymce-templates/image-grid-2x2.html"
  },
  {
   "title": "Content box",
   "url": "++theme++plonetheme.tango/tinymce-templates/content-box.html"
  }
]
```

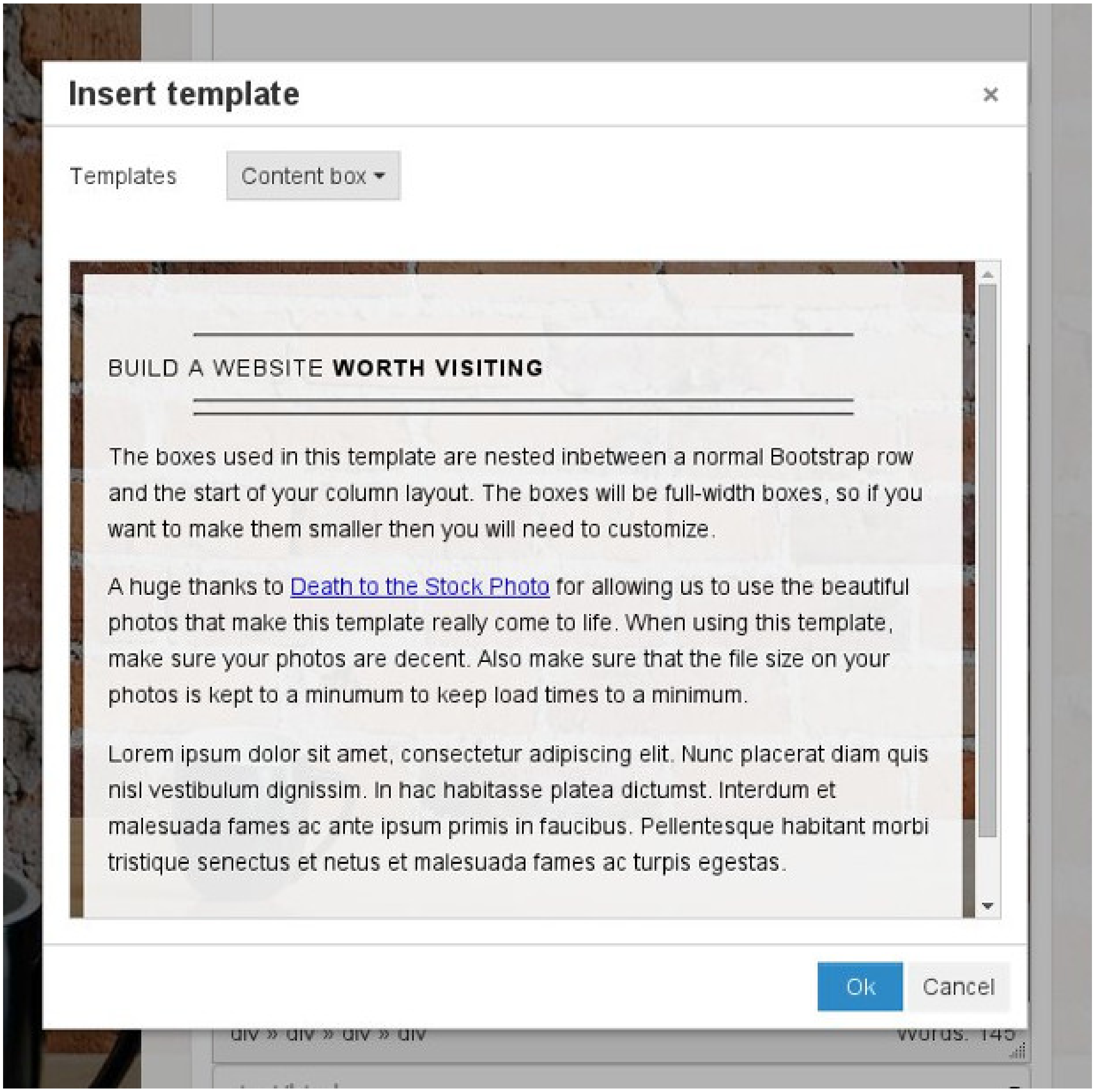
Note: Just remember to activate the plugin from the toolbar/plugins tab.

Use TinyMCE templates for content creation

We can add template-based content from the *Insert* menu > *Insert template*:



Now we can choose one of the existing TinyMCE templates:



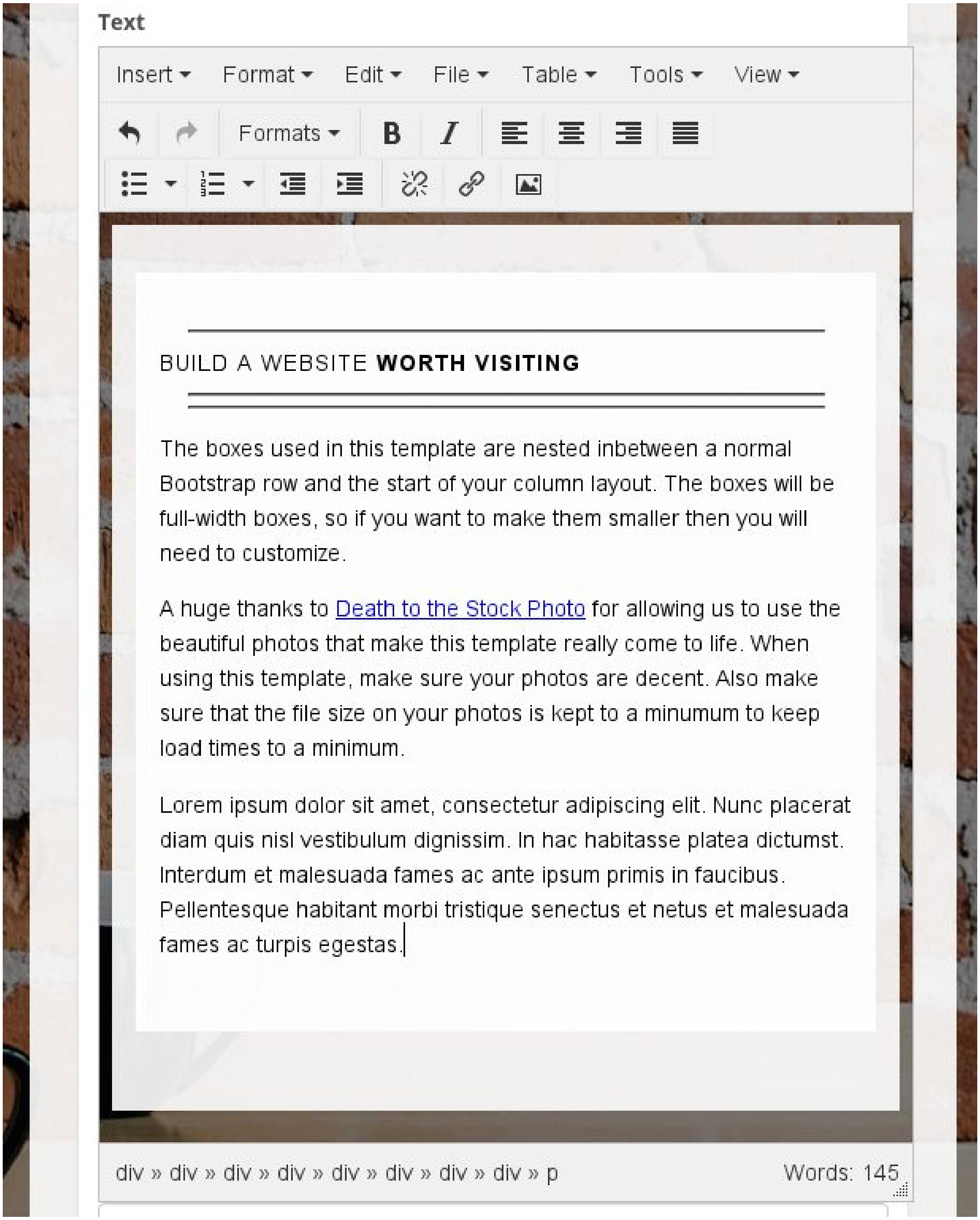
BUILD A WEBSITE **WORTH VISITING**

The boxes used in this template are nested inbetween a normal Bootstrap row and the start of your column layout. The boxes will be full-width boxes, so if you want to make them smaller then you will need to customize.

A huge thanks to Death to the Stock Photo for allowing us to use the beautiful photos that make this template really come to life. When using this template, make sure your photos are decent. Also make sure that the file size on your photos is kept to a minumum to keep load times to a minimum.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nunc placerat diam quis nisl vestibulum dignissim. In hac habitasse platea dictumst. Interdum et malesuada fames ac ante ipsum primis in faucibus. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas.

After we have chosen our template and then clicked on *OK*, we have our template-based content in the editor:



BUILD A WEBSITE **WORTH VISITING**

The boxes used in this template are nested inbetween a normal Bootstrap row and the start of your column layout. The boxes will be full-width boxes, so if you want to make them smaller then you will need to customize.

A huge thanks to Death to the Stock Photo for allowing us to use the beautiful photos that make this template really come to life. When using this template, make sure your photos are decent. Also make sure that the file size on your photos is kept to a minumum to keep load times to a minimum.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nunc placerat diam quis nisl vestibulum dignissim. In hac habitasse platea dictumst. Interdum et malesuada fames ac ante ipsum primis in faucibus. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas.

div » div » div » div » div » div » div » div » p

Words: 145

We can now customize it or use more templates to create more content.

Advanced resources registry usage

Note: For theming in general you don't need to use the resource registry. The following infos stay here only as an example usage of the resource registry.

In the Plone resource registry we can register our static resources, like CSS and LESS files and also JavaScript resources. In fact even our resources we defined in the manifest.xml are registered here automatically, but hidden. We will cover here only CSS and LESS, but you can also do nice things with your JavaScript resources (for example using requirejs to do the import correctly without worrying about import order). For details about this, look into the documentation of the resource registry and in the JavaScript part of the training.

Registering CSS/Less resources in the registry

Because of the flexibility of Less over CSS we will use only Less files here, but static CSS files can be registered in the same way. Less files have the advantage that we can use imports, and with reference-imports we can even import only the parts of the files which we are really using.

Let's see how we can register a resource in the resource registry. To do that, we add an IResourceRegistry entry into the registry.xml in our profiles/default folder:

```
<?xml version="1.0"?>
<registry>
    <records prefix="plone.resources/tango-main"
              interface='Products.CMFPlone.interfaces.IResourceRegistry'>
       <value key="css">
          <element>++theme++plonetheme.tango/css/main.less</element>
       </value>
    </records>
</registry>
```

This registers a file named main.less (from our theme package named plonetheme.tango) as a *resource* named tango-main. We can now add this resource to a *resource bundle* like the existing plone bundle:

```
<?xml version="1.0"?>
<registry>
    <records prefix="plone.resources/tango-main"
              interface='Products.CMFPlone.interfaces.IResourceRegistry'>
       <value key="css">
          <element>++theme++plonetheme.tango/css/main.less</element>
       </value>
    </records>

    <records prefix="plone.bundles/plone"
              interface='Products.CMFPlone.interfaces.IBundleRegistry'>
      <value key="resources" purge="false">
        <element>tango-main</element>
      </value>
    </records>
</registry>
```

This has the advantage of reducing the number of bundles, which also means reducing the amount of files which are loaded for the site, because every bundle will result in *one* compiled CSS file and *one* compiled JavaScript file. So if we have multiple LESS resources in the same bundle, they will be merged into one compiled CSS file.

We can also create our own custom bundle which contains our resource:

```
<?xml version="1.0"?>
<registry>
    <records prefix="plone.resources/tango-main"
              interface='Products.CMFPlone.interfaces.IResourceRegistry'>
       <value key="css">
          <element>++theme++plonetheme.tango/css/main.less</element>
       </value>
    </records>

    <!-- bundle definition -->
    <records prefix="plone.bundles/tango-bundle"
              interface='Products.CMFPlone.interfaces.IBundleRegistry'>
      <value key="resources">
        <element>tango-main</element>
      </value>
      <value key="enabled">True</value>
      <value key="compile">True</value>
      <value key="csscompilation">++theme++plonetheme.tango/css/tango-compiled.css</
↪value>
      <value key="last_compilation"></value>
    </records>
</registry>
```

This can make sense if we only want to load that bundle under certain conditions, like in a specific context. This could lead to a smaller size of loaded static resources, when they are not all needed.

After making changes to the registry, like adding resources to a bundle, you have to reload the registry configuration via an upgrade step, or via a uninstall/install of the package.

If you change a bundle, it has to be built or rebuilt. You can do this in the `@@resourceregistry-controlpanel` by clicking on *Build* for the bundle involved.

Advanced Diazo

“Diazo allows you to apply a theme contained in a static HTML web page to a dynamic website created using any server-side technology.”

To do this, Diazo does some real complicated stuff on your behalf: it writes XSLT!

But sometimes basic rules are not enough and you need to write a bit of XLST yourself.

Modify the theme and the content on the fly

Let’s look at some examples from the official diazo docs.

Extend rules

You can re-use or extend rules from another theme or from another file in your theme.

A good example of a use case is the one described by Asko Soukka (thanks!!!) in this blog post about how to Customize Plone 5 default theme on the fly.

Include external content

You can include external content from another website or from a custom view.

Recipes and snippets

The docs provide a basic recipe set and you can have your own, but how to remember and re-use them?

David Bain introduces a “diazo snippets library” that allows you to get snippets from a chrome extensions. All the snippets are available here.

More snippets

Make some links open in new window

```
<!-- add target="_blank" to all links in portlet-collection-links -->
<xsl:template match="//dl[contains(@class,'portlet-collection-links')]//a">
  <a target="_blank"><xsl:apply-templates select="./@*[contains(' href title class rel ', concat(' ', name(), ' '))]"/><xsl:value-of select="." /></a>
</xsl:template>
```

At diazo.org is another way described in the recipes: http://docs.diazo.org/en/latest/recipes/adding-an-attribute/index.html

Add CSS marker classes depending on existing `portal-columns`

This adds a CSS class for every existing `portal-column` to the `body` tag. If `portal-column-one` exists, we add `col-one`; if `portal-column-content` exists, we add `col-content`; and if `portal-column-two` exists, we add `col-two`.

```
<before theme-children="/html/body" method="raw">
  <xsl:attribute name="class">
    <xsl:value-of select="/html/body/@class" />
    <xsl:if css:test="#portal-column-one"> col-one</xsl:if>
    <xsl:if css:test="#portal-column-content"> col-content</xsl:if>
    <xsl:if css:test="#portal-column-two"> col-two</xsl:if>
  </xsl:attribute>
</before>
```

Now, one can use these markers to define the grid in a semantic way like this:

```
body.col-one.col-content.col-two #content-wrapper {
  .make-row();

  #portal-column-content {
    .make-lg-column(6);
    .make-lg-column-offset(3);
  }

  #portal-column-one {
    .make-lg-column(3);
    .make-lg-column-pull(9);
  }
  #portal-column-two {
    .make-lg-column(3);
  }
}
body.col-content #content-wrapper {
  .make-row();

  #portal-column-content {
    .make-lg-column(12);
  }
}
```

Note: This way, you don't need the xsl-rules Barceloneta uses to create the main content area. It's more flexible than Barceloneta's approach. Another way could be, to change Plone to provide these classes already ;).

Move Plone elements around

Sometimes one needs to move Plone elements from one place to another or merge some elements together. In the following example we merge the language flags together with the document actions.

```
<replace css:content-children=".documentActions > ul">
  <xsl:for-each select="//*[@class='documentActions']/ul/li">
    <xsl:copy-of select="." />
  </xsl:for-each>
  <xsl:for-each select="//*[@id='portal-languageselector']/*">
    <xsl:copy-of select="." />
  </xsl:for-each>
</replace>
```

Taking over specific portlets

```
<!-- all portal-column-two portlets but not portletNews and not portletEvents -->
<after
  content="//div[@id='portal-column-two']//dl[not(contains(@class,'portletNews')) and␣
↪not(contains(@class,'portletEvents'))]"
  css:theme-children="#portal-column-two"
  />
```

```
<!-- all portal-column-one portlets but not portletNavigationTree -->
<after
  content="//div[@id='portal-column-one']//dl[not(contains(@class,
↪'portletNavigationTree'))]"
  css:theme-children='#portal-column-two'
  />
```

Customize template on the fly: collective.jbot

> **Warning:** Not yet compatible with Plone 5!

Back in the old days we used to customize views' templates from the tool *portal_view_customizations* from the ZMI.

This tool has no UI and could beat you whenever you don't expect it.

So, Nathan Van Gheem has created this package that unfortunately is not (yet!) part of the core: collective.jbot.

This package uses the well-known *z3c.jbot* under the hood and allows you to customize templates TTW from the control panel.

Installation

TODO

Usage

TODO

SCREENSHOTS

Additional goodie

Overrides are stored on the filesystem and you can version / backup them as you like!

Creating custom components

Plone is a very powerful system and it provides many interesting things for you. To dive into this, we recommend to go thru the `Mastering Plone 5: Development` of the training.

For theming the most relevant part are the following components, which render some parts of Plone, you may want to customize or build new once.

Views

In Plone a view usually consists of multiple components, a Python class based on BrowserView and a template which renders the markup. You as you have already seen, you can also have template only iews. It is also possible to have a view which has no template, but renders the output by it self, as JSON for example.

For more details about views and there possibilities see the view sections of the `Mastering Plone 5: Development` chapters.

Viewlets

Viewlets are small pieces which are rendered inside a view. The are registered for a ordered ViewletManager, which renders all Viewlets in the given order. You can change the order even TTW or via configuration. A Viewlet consists of a Viewlet Python class and a template. Plone many default Viewlets and ViewletManagers like ContentAbove and BelowContent which you can use to register small pieces of functionality.

For an overview of existing Viewlets and ViewletManagers look at the `/@@manage-viewlets` view.

For more details about Viewlets/ViewletManagers and there possibilities see the Viewlets sections of the `Mastering Plone 5: Development` chapters.

Portlets

Portlets are a very flexible way of providing context related information in the right, left or footer area.

For details on how to use, configure and create Portlets, look into the Plone docs Plone docs Portlet sections

See also:

- *"Through-the-web" Plone customization*
- http://docs.plone.org/adapt-and-extend/theming/index.html

JavaScript for Plone Developers

The definitive location for documentation regarding Plone's JavaScript and Resource Registries is located at: http://docs.plone.org/adapt-and-extend/theming/resourceregistry.html

Training Objective

The most important objective of this training is mostly to explain how to integrate JavaScript applications and integrations into Plone in many different scenarios. Secondly, it is to explain the JavaScript technologies used in Plone itself(RequireJS, Patterns, Resource registry).

This training is *not* about:

- how to write JavaScript
- how to write React/Angular 2/JS framework of the week

Sections

The JavaScript development process in Plone

Code style

Together with `plone.api` we developed code style guidelines, which we are enforcing now for core Plone development. Finally! This makes code so much more readable. It currently doesn't cover JavaScript code guidelines, but those were considered when Mockup was developed. And luckily, similar to PEP 8 and the associated tooling (**`pep8`**, **`pyflakes`**, **`flake8`**), JavaScript also has some guidelines - not official, but well respected. Douglas Crockford - besides of specifying the JSON standard - wrote the well known book "JavaScript the good parts". Out of that he developed the code linter JSLint. Because this one was too strict, some other people wrote JSHint.

Mockup uses JSHint with the following .jshintrc configuration file:

```
{
   "bitwise": true,
   "curly": true,
   "eqeqeq": true,
   "immed": true,
   "latedef": true,
   "newcap": true,
   "noarg": true,
   "noempty": true,
   "nonew": true,
   "plusplus": true,
   "undef": true,
   "strict": true,
   "trailing": true,
   "browser": true,
   "evil": true,
   "globals": {
      "console": true,
      "it": true,
      "describe": true,
      "afterEach": true,
      "beforeEach": true,
      "define": false,
      "requirejs": true,
      "require": false,
      "tinymce": true,
      "document": false,
      "window": false
   }
}
```

Note: When working with JSHint or JSLint, it can be very useful to get some more context and explanation about several lint-errors. For JSHint there is a list of all configurable options: http://jshint.com/docs/options/

We strongly recommend to configure your editor of choice to do JavaScript code linting on save. The Mockup project is enforcing Lint-error-free code. Besides of that, this will also make you a better coder. The JSHint site lists some editors with Plugins to support JSHint linting: http://jshint.com/install/

Regarding spaces/tabs and indentation:

- Spaces instead of tabs.
- Tab indentation: 2 characters (to save screen estate).

You have to configure your editor to respect these settings.

Confirming on a common code style makes contributing much more easier, friendly and fun!

Mockup contributions

For each feature, create a branch and make pull-requests on Github. Try to include all your changes in one commit only, so that our commit history stays clean. Still, you can do many commits to not accidentally loose changes and still commit to the last commit by doing:

```
git commit --amend -am"my commit message".
```

Don't forget to also include a change log entry in the `CHANGES.rst` file.

Documentation

Besides documenting your changes in the `CHANGES.rst` file, also include user and developer documentation as appropriate.

For patterns, the user documentation is included in a comment in the header of the pattern file, as described in *Writing documentation for Mockup*.

For function and methods, write an API documentation, following the apidocjs standard. You can find some examples throughout the source code.

We also very welcome contributions to the training documentation and the official documentation. As with other contributions: please create branches and make pull-requests!

RequireJS and JavaScript modules

One of the great new features, Plone 5 gives us, is the ability to define and use JavaScript modules.

Most serious programming languages provide the concept of namespaces and module dependencies, like Python's `import` mechanism. Python code would be unmanageable, if we'd rely on the existence of global variables and objects in our own scripts.

But JavaScript doesn't have any concept for declaring dependencies. Only the new and finalized ECMAScript 6 (ES6) standard finally comes with a module definition system (actually directly inspired by RequireJS and CommonJS), along other great features like proper variable scoping.

In Plone, we use RequireJS as a framework to define and load modules. RequireJS is an implementation of the Asynchronous Module Definition API. The module definition and loading standard of CommonJS is used by NodeJS. RequireJS adds the ability to load modules asynchronously, which can be better for performance. The CommonJS module loading syntax can also be used in RequireJS. But the main point why Plone uses RequireJS is, that there is a JavaScript based compiler, which allows us to build bundles (a combined, optimized and minified form with all dependencies) Through-The-Web. RequireJS and CommonJS are also forward compatible with ES6's module definition standard.

Finally we can use JavaScript in Plone like it is a proper programming language! No need to depend on the existence of global variables and a strict order, in which scripts have to be loaded. You can still use legacy-style JavaScript, but Plone encourages you to enter the modern world of JavaScript development.

Defining a module

In the past years, a common pattern of defining anonymous function calls has evolved. This allows to better scope variables and not clutter the global namespace. The pattern is discussed in depth at JavaScript Module Pattern: In-Depth and basically comes down to the following Pattern:

```
(function ($, _) {
    // now have access to globals jQuery (as $) and underscore (as _) in this code.
}(jQuery, underscore));
```

If your code should be reused like a library, you can define a module export.

```
var my_module = (function ($, _) {
    var ret = {};
    ret.my_method = function () {
        // do something
    }
    return ret;
}(jQuery, underscore));
```

RequireJS extends this pattern and removes the necessity for globals to refer to other modules. In RequireJS, you're wrapping your code like this:

```
define(["jquery", "underscore"], function($, _) {
    // now have access to jQuery (as $) and underscore (as _), both defined as modules in the RequireJS configuration.
    var ret = {};
    ret.my_method = function () {
        // do something
    }
    return ret;
});
```

No need for any globals anymore (except for the `define` and `require` methods)!

Also note, that the code within the RequireJS define wrapper is exactly the same as in the module pattern example above. Using RequireJS doesn't mean, you have to rewrite everything. It's just about modularizing code.

To be able to use the defined module somewhere else, you need to be able to reference it by a module id. You can pass it as very first argument to the `define` function, but you might better do that in the RequireJS configuration. If you don't do it at all, it gets automatically assigned the name of the file. For example, let's assume a project structure like follows and the `define` example from above living in a file called `my_module.js`:

```
index.html
require.js
my_project/
        |___main.js
        |___app/
              |___/my_module.js
```

Let's do the RequireJS configuration in `main.js` and use that as main entry point also to finally let something happen:

```
require.config({
  baseUrl: "my_project/",
  paths: {
      "app": "app/"
  }
});
require(['app/my_module'], function (my_module) {
    my_module.my_method();
})
```

You can use your defined module as a dependency in another `define` module definition - if you just want to run some non-reusable code - as a dependency in a `require` call. While you have to return a module export in `define`, you don't need that for `require`. So, `require` corresponds to the first form of the module pattern explained above.

When using in the browser (and not in NodeJS, for example), we have to include an entry point as script tag in our HTML markup:

```
<script src="require.js"></script>
<script src="my_module/main.js"></script>
```

Alternatively, you can define a script as main entry point in RequireJS as data attribute on the script tag, which loads require.js. In that case, you could omit the configuration, because the entry point script is used as `baseUrl`, if nothing else is defined:

```
<script data-main="my_project/main.js" src="require.js"></script>
```

More information

More on RequireJS' API and how to include legacy code, which doesn't use the `define` module definition pattern, see the RequireJS API documentation.

Mockup - A Patternslib based collection of components

Plone 5 ships with a revamped UI. An important part of the new UI is a collection of new input widgets, that we are used to work with in the so called Web 2.0 era.

For Plone, it was high time to update and modernize its input widgets. Not because the new ones look much better, but because they offer a much more comfortable way of entering data. To update Plone's widgets was the goal of `plone.app.widgets`, started by Nathan van Gheem and pushed wide forward by Rok Garbas. Rok forked Patternslib and created the Mockup project. Patternslib used a complex configuration syntax parser instead of a simple JSON based approach and the test coverage was not high enough. Besides it was fun to create something new, so Mockup was born. There were concerns about having two projects with the same goal, so JC Brand took the initiative and brought the two projects back together. Where Mockup had a dependency on `mockup-core` with a base pattern to extend from, a configuration parser, pattern registry and Grunt infrastructure, this dependency was removed and replaced by a dependency on `patternslib`.

Those projects led the foundation to the new way of developing JavaScript in Plone.

The Mockup project structure

This is how Mockup is structured on the filesystem:

```
mockup
-- bower.json                - Bower managed dependencies
-- Makefile                  - Makefile targets to bootstrap, build bundles
-- mockup                    - All the source in here (to be able to add to Python egg)
|   -- bower_components      - External dependencies managed by Bower
|   -- configure.zcml        - Registers Mockup resource directory
|   -- Gruntfile.js          - Grunt build configuration. Extends js/grunt.js
|   -- index.html            - Entry file for generated documentation
|   -- js
|   |   -- bundles           - Mockup bundle files
|   |   |   -- docs.js
|   |   |   -- plone.js
|   |   |   -- widgets.js
|   |   |   -- ...
|   |   -- config.js         - RequireJS configuration
|   |   -- docs              - ReactJS based documentation framework
|   |   |   -- app.js
|   |   |   -- ...
|   |   -- grunt.js          - Grunt base configuration
|   |   -- i18n.js
|   |   -- i18n-wrapper.js
|   |   -- router.js
|   |   -- ui
|   |   |   -- templates
|   |   |   |   -- popover.xml
|   |   |   -- views
|   |   |       -- base.js
|   |   |       -- buttongroup.js
|   |   |       -- ...
|   |   -- utils.js          - Utils to be reused
|   -- less                 - Less files for bundles. Mostly import less files from
|   |   -- base.less          a bundle's pattern dependencies.
|   |   -- docs.less
|   |   -- plone.less
|   |   -- widgets.less
|   |   -- ...
|   -- lib                                 - Non-Bower libraries
|   |   -- jquery.event.drag.js
|   |   -- jquery.event.drop.js
|   -- node_modules -> ../node_modules     - Grunt needs this link here...
|   -- patterns                            - Patterns in here
|   |   -- autotoc                         - The autodoc pattern
|   |   |   -- pattern.autotoc.less        - Pattern specific Less file
|   |   |   -- pattern.js                  - Pattern itself
|   |   -- livesearch
|   |   |   -- pattern.js
|   |   |   -- pattern.livesearch.less
|   |   -- select2
|   |   |   -- pattern.js
|   |   |   -- pattern.select2.less
|   -- tests                               - All tests in here
|       -- config.js                       - RequireJS configuration for tests
|       -- fakeserver.js                   - Fake test server
|       -- files
|       |   -- lessconfig.js
|       |   -- mapper.html
|       |   -- r.js
|       -- i18n-test.js
|       -- images                          - Test resources
|       |   -- extralarge.jpg
|       |   -- large.jpg
|       |   -- ...
|       -- json                            - Test data
|       |   -- contextInfo.json
|       |   -- fileTree.json
|       |   -- ...
|       -- pattern-autotoc-test.js         - Tests for the autodoc pattern
|       -- pattern-livesearch-test.js
|       -- pattern-select2-test.js
|       -- ...
-- node_modules             - Node modules directory
-- package.json             - Node package metadata
-- provision.sh             - Vagrant provision file
-- setup.py                 - Mockup egg setup
-- Vagrantfile              - Vagrant configuration
```

A minimal pattern

The following is a minimal pattern example, except that it uses jQuery and changes some HTML elements' text.

```
define([
  'pat-base',
  'jquery'
], function (Base, $) {
  'use strict';

  var Minimalpattern = Base.extend({
    name: 'minimalpattern',
    trigger: '.pat-minimalpattern',  // has to be exact like this: 'pat-' +␣
↪patternname.
    defaults: {                      // default options
      text: 'Super Duper!'
    },
    init: function () {              // pattern initialization. called for each␣
↪matching pattern.
      var self = this;
      self.$el.html(self.options.text);  // self.$el is the matching pattern element.
    }
  });
  return Minimalpattern;
});
```

For a complete example including tests, bundle config und Plone integration see: https://github.com/collective/mockup-minimalpattern

Writing documentation for Mockup

The documentation for Mockup is automatically generated from comments in pattern code. The structure is as follows:

```
/* PATTERN TITLE
 *
 * Options:
 *    OPTION_TITLE(TYPE): DESCRIPTION
 *    OPTION2_TITLE(TYPE): DESCRIPTION2
 *
 * Documentation:
 *   # Markdown title
 *
 *   Markdown structured description text
 *
 *   # Example
 *
 *   {{ EXAMPLE_ANCHOR }}
 *
 *   # Example2
 *
 *   {{ EXAMPLE2_ANCHOR }}
 *
 * Example: EXAMPLE_ANCHOR
 *    <div class="pat-PATTERN_NAME"></div>
 *
 * Example2: EXAMPLE2_ANCHOR
 *    <section class="pat-PATTERN_NAME"></section>
 *
 * License:
 *   License text, if it differs from the package's license, which is
 *   declared in package.json.
 *
 */
```

Through-The-Web development

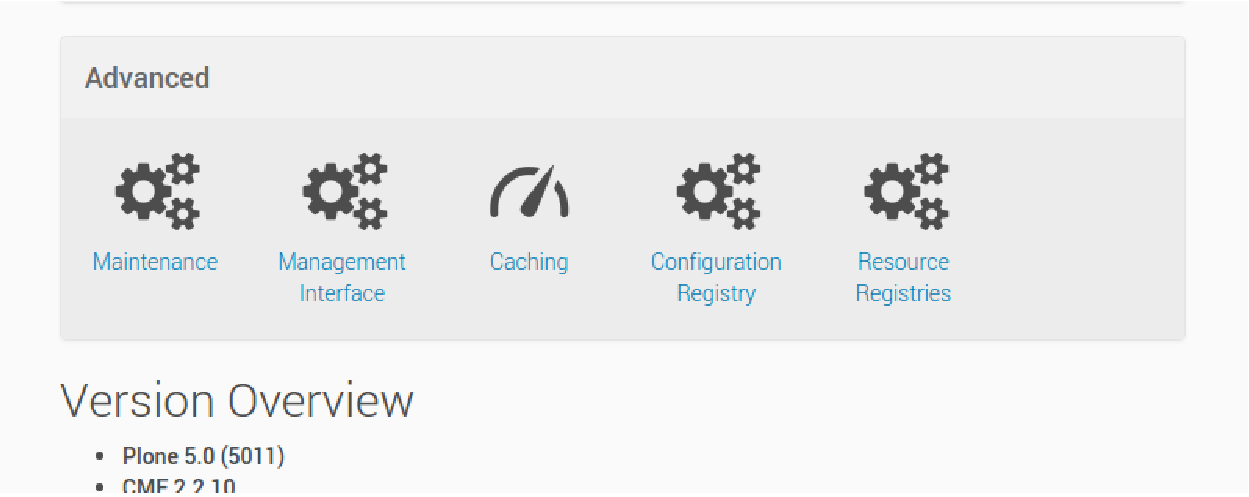
It is possible to include Javascript functionality without the need to know about any of the tools involved. This is not reccommended for when you need to do a complex and modular implementation.

portal_javascript & portal_css

These two portal tools are no longer used in Plone 5. They are still present, but nothing should be included in them.

Resource Registries

This is the new tool included in Plone 5. From here we will manage everything related to Javascript and CSS resources. It can be found right at the bottom of Plone's Control Panel, in the *Advanced* section.



Add files

We are going to include 2 new resources, a Javascript file, and a LESS file.

The Javascript will look like this:

```
$( document ).ready(function() {
    var links = $('a');
    links.addClass('custom-background');
});
```

The LESS will look like this:

```
a.custom-background{
    background-color: #F7E1CF;
    color: black;
}
```

- Go to the *Overrides* tab
- Click the *Add file* button
- Name the new file `++plone++static/custom-links.js`

- Paste the contents of the Javascript section into the textarea
- Click *Save*
- Click the *Add file* button again
- Name the new file `++plone++static/custom-links.less`
- Paste the contents of the CSS section into the textarea
- Click *Save*

Create the resource

- Go to the *Registry* tab
- Click the *Add resource* button
- Name it `training-custom-links`
- Under `JS` enter `++plone++static/custom-links.js`
- For the *CSS/LESS* section, click *Add*
- Enter `++plone++static/custom-links.less`

It should look somthing like this:

- Click *Save*

Create the bundle and wire everything up

- Go to the *Registry* tab
- Click the *Add bundle* button
- Name it `training-custom-bundle`
- Under *Resources* enter `training-custom-links`
- For the *Depends* section, we'll use `plone`
- Make sure *Enabled* is checked

It should look somthing like this:

- Click *Save*

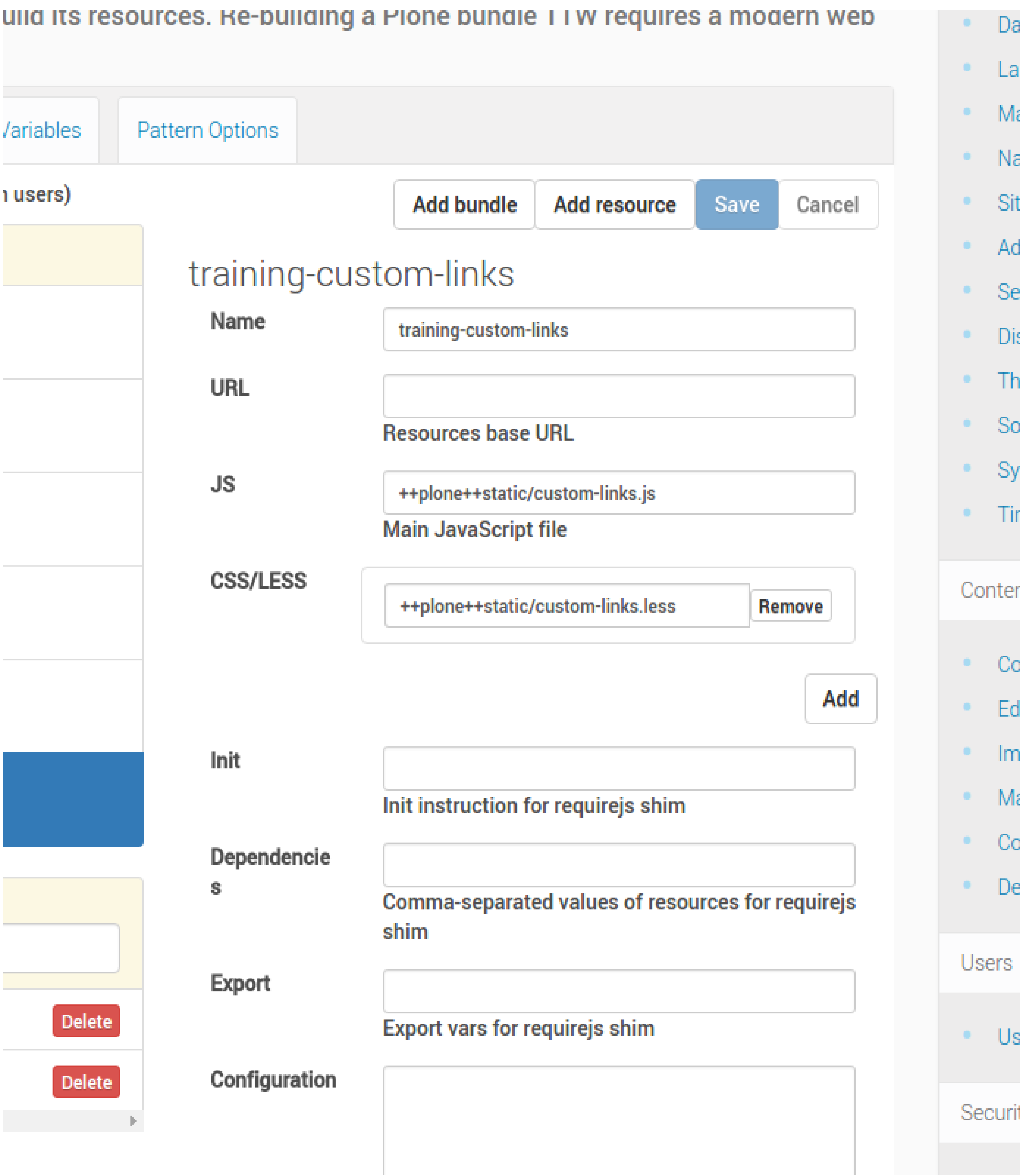
Build the bundle

In order for changes to be included, you need to build your bundle. For doing this, you just need to click the *Build* under the bundle you want to build.

Exercises

Prerequisites

- Follow the instructions here to get a training buildout installed: https://training.plone.org/5/plone_training_config/instructions.html



training-custom-links

Name: training-custom-links

URL:

Resources base URL

JS: ++plone++static/custom-links.js

Main JavaScript file

CSS/LESS: ++plone++static/custom-links.less

Init:

Init instruction for requirejs shim

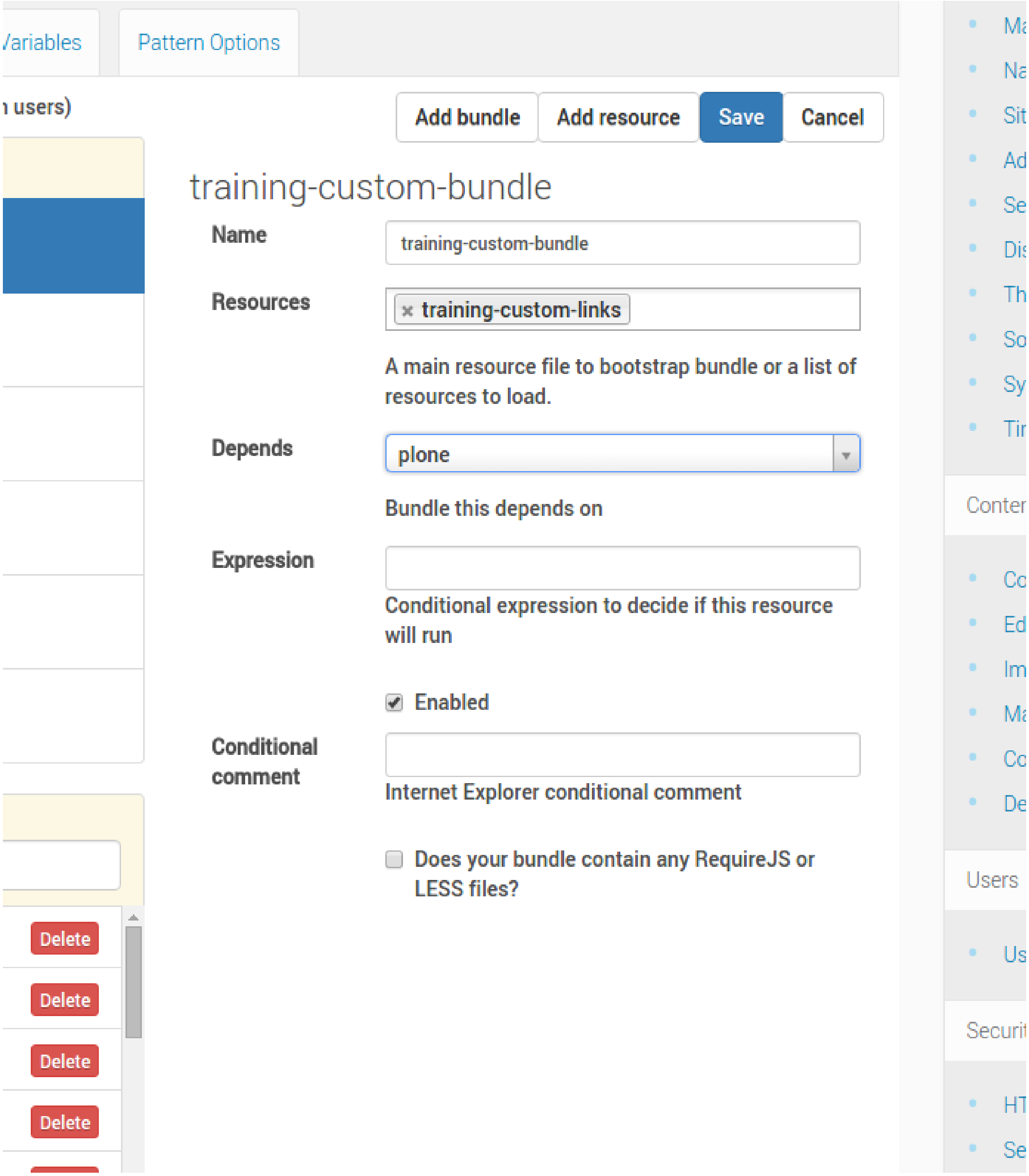
Dependencies:

Comma-separated values of resources for requirejs shim

Export:

Export vars for requirejs shim

Configuration:



training-custom-bundle

Name: training-custom-bundle

Resources: × training-custom-links

A main resource file to bootstrap bundle or a list of resources to load.

Depends: plone

Bundle this depends on

Expression:

Conditional expression to decide if this resource will run

☑ Enabled

Conditional comment:

Internet Explorer conditional comment

□ Does your bundle contain any RequireJS or LESS files?

- Fork https://github.com/collective/collective.jstraining and install your fork into your buildout from the previous step
- npm/nodejs install on your system
- webpack installed on your system
- grunt-cli installed on your system(`npm install -g grunt-cli`)

Install forked collective.jstraining

Add this line to the end of your `buildout.cfg` file:

> collective.jstraining = git <location of your fork>

`<location of your fork>` should be replaced with where your fork is.

Also, add `collective.jstraining` to the auto-checkout list:

```
auto-checkout =
  ...
  collective.jstraining
  ...
```

And one more spot to add collective.jstraining to: eggs:

```
eggs =
    ...
    collective.jstraining
    ...
```

Exercises

Exercise 1: Include JavaScript in browser view

> **Warning:** This exercise requires a working buildout using a fork of the collective.jstraining package.

For this exercise, we are simply including JavaScript in a browser view.

We will be working in the `exercise1` directory of the collective.jstraining package.

Add your JavaScript file

First off, in your `exercise1/static` directory, add a file named `script.js`. This exercise is open ended as to what you do with JavaScript on the page. We'll stay very simple for the sake of brevity, using jQuery to do a simple animation effect on the title of the page:

```
require([
  'jquery'
], function($){

  var cycle = function(){
    $('h1').animate({
      left: '250px',
      opacity: '0.5',
      'font-size': '30px'
    }, function(){
      $('h1').animate({
        left: '0',
        opacity: '1',
        'font-size': '20px'
      }, function(){
        setTimeout(function(){
          cycle();
        }, 2000);
      });
    });
  };

  $(document).ready(function(){
    cycle();
  });
});
```

Feel free to customize the script to do whatever you'd like.

Register static resource directory

Next, let's register the static directory we just placed our script into. To register, you need to add ZCML registration for the static directory your script is in. Add this to the `exercise1/configure.zcml` file:

```
<plone:static
     directory="static"
     type="plone"
     name="exercise1"
     />
```

Register JavaScript resource

Let's register our script as a JavaScript resource with Plone. In the `exercise1/profiles/default/registry.xml` file, add configuration to register your script:

```
<records prefix="plone.resources/exercise1"
              interface='Products.CMFPlone.interfaces.IResourceRegistry'>
      <value key="js">++plone++exercise1/script.js</value>
  </records>
```

Create your browser view

Warning: This might be redundant with other documentation. Skip ahead if you know how to create browser views.

Finally, let's load our JavaScript file to only load on a specific page you need it on.

In our case, let's add a basic new page view. The page template doesn't need to implement any logic and we can use the main template to bring in the content of the page we're using in the JavaScript(h1). Add this into your `exercise1/page.pt` file:

```
<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="context/main_template/macros/master"
    i18n:domain="plone">
<body>

</body>
</html>
```

Load your JavaScript resource

Add in view python code to tell Plone to render the script in the `exercise1/browser.py` file:

```
from Products.CMFPlone.resources import add_resource_on_request
from Products.Five import BrowserView


class Exercise1View(BrowserView):

    def __call__(self):
        # utility function to add resource to rendered page
        add_resource_on_request(self.request, 'exercise1')
        return super(Exercise1View, self).__call__()
```

The most interesting part here is to look at `add_resource_on_request`.

Finally, wire it up with ZCML registration in the `exercise1/configure.zcml` file:

```
<browser:page
     name="exercise1"
     for="*"
     class=".browser.Exercise1View"
     template="page.pt"
     permission="zope2.View"
     />
```

Installation

1. Start up your Plone instance
2. Install the `Exercise 1` add-on

Then, visit the URL: `http://localhost:8080/Plone/front-page/@@exercise1`. This is assuming your Plone is is located at the URL `http://localhost:8080/Plone`.

Production

In this exercise, there is no special distinction between development and production builds. The JavaScript is developed without any build process.

Exercise 2: NG2 APP component rendered in a browser view

Warning: This exercise requires a working buildout using a fork of the collective.jstraining package.

For this exercise, we will run an angular 2 application inside a plone browser view.

We have most of the angular 2 boiler plate code created for you so let's just finish up a few things so you can customize it.

In this case we are going to use angular client to create the app inside the package.

We will be working in the `exercise2` directory of the collective.jstraining package.

Bootstrap

Install npm dependencies:

```
cd exercise2/static/ng2app
npm install
npm install -g angular-cli
```

Add your angular 2 component

In your `exercise2/static/ng2app` directory, there is a bolierplate code for an ng2 app. You can use ng2 cli to create new components, modules, services,... We hope you like typescript.

We can change the exercise2/static/src/app/app.component.html to create your own template.

Like I said, you can do whatever in this module.

Register static resource directory

Next, let's register the static directory we just placed our script into. To register, you need to add ZCML registration for the static directory your script is in. Add this to the `exercise2/configure.zcml` file:

```
<plone:static
     directory="static"
     type="plone"
     name="exercise2"
     />
```

Build the file with webpack

Our deployment is built using the ng cli tool:

```
cd exercise2/static/ng2app
ng build --prod
```

Whenever you make a change to your component files, webpack will auto re-build the distribution

Register JavaScript resource

Angular CLI creates three js, one for basic webpack instructions, one with the main js and another with the styling js. You will need to register the three on the `exercise2/profiles/default/registry.xml`:

```
<records prefix="plone.resources/exercise2-inline"
          interface='Products.CMFPlone.interfaces.IResourceRegistry'>
  <value key="js">++plone++exercise2/ng2app/dist/inline.js</value>
</records>
<records prefix="plone.resources/exercise2-main"
          interface='Products.CMFPlone.interfaces.IResourceRegistry'>
  <value key="js">++plone++exercise2/ng2app/dist/main.8b778eea5dd35968ef66.bundle.js</
↪value>
  <value key="deps">exercise2-inline</value>
  <value key="deps">exercise2-style</value>
</records>
<records prefix="plone.resources/exercise2-style"
          interface='Products.CMFPlone.interfaces.IResourceRegistry'>
  <value key="js">++plone++exercise2/ng2app/dist/styles.b52d2076048963e7cbfd.bundle.js
↪</value>
</records>
```

Its really important that in case that you need to have dependency on loading the js you define on the registry.xml as its showen for the main js.

Finally we want to create a single entry point to load them, so we are going to create and register a js with the requires that are loading the app on a file called `static/ng2app/main.js`:

```
require(['exercise2-inline','exercise2-style','exercise2-main'])
```

With the main.js defined on the filesystem we can now create the resource as a new resource:

```
<records prefix="plone.resources/exercise2"
          interface='Products.CMFPlone.interfaces.IResourceRegistry'>
  <value key="js">++plone++exercise2/ng2app/main.js</value>
</records>
```

Create your browser view

Warning: This might be redundant with other documentation. Skip ahead if you know how to create browser views.

Finally, let's load our JavaScript file to only load on a specific page you need it on.

In our case, let's add a basic new page view. The page template doesn't need to implement any logic and we can use the main template to bring in the content of the page we're using in the JavaScript(h1). Add this into your `exercise2/page.pt` file:

```
<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="context/main_template/macros/master"
    i18n:domain="plone">
<body>

  <metal:content-core fill-slot="content-core">
  <metal:content-core define-macro="content-core">
    <app-root></app-root>
  </metal:content-core>
  </metal:content-core>

</body>
</html>
```

The `app-root` tag is what is used for the component selector. You can customize this and use whatever selector you like.

Load your JavaScript resource

Add in view python code to tell Plone to render the script in the `exercise2/browser.py` file:

```python
from Products.CMFPlone.resources import add_resource_on_request
from Products.Five import BrowserView


class Exercise2View(BrowserView):

    def __call__(self):
        # utility function to add resource to rendered page
        add_resource_on_request(self.request, 'exercise2')
        return super(Exercise2View, self).__call__()
```

The most interesting part here is to look at `add_resource_on_request`.

Finally, wire it up with ZCML registration in the `exercise2/configure.zcml` file:

```xml
<browser:page
     name="exercise2"
     for="*"
     class=".browser.Exercise2View"
     template="page.pt"
     permission="zope2.View"
     />
```

Installation

1. Start up your Plone instance
2. Install the `Exercise 2` add-on
3. Toggle development mode to make sure the new resources are included

Then, visit the URL: `http://localhost:8080/Plone/front-page/@@exercise2`. This is assuming your Plone is is located at the URL `http://localhost:8080/Plone`.

Warning: To make sure your resource registry configuration changes apply, you'll need to be in development mode. You can also toggle development mode on and off, click save, to force configuration to be re-built after changes instead of keeping development mode on.

Production

In this exercise, there is no special distinction between development and production builds. Webpack re-builds the resource on every change for you and the JavaScript build file is not added to any bundle–it is just loaded for this particular page.

Exercise 3: NG2 APP component in a bundle

Warning: This exercise requires a working buildout using a fork of the collective.jstraining package.

For this exercise, we will add an angular 2 application to a plone bundle.

We have most of the angular 2 boiler plate code created for you so let's just finish up a few things so you can customize it.

We will be working in the `exercise3` directory of the collective.jstraining package.

Bootstrap

Install npm dependencies:

```
cd exercise3/static
npm install
```

Add your angular 2 component

In your `exercise3/static/app` directory, add a file named `app.component.ts`. Use this file to do anything you would like to the page. This example will stick with the angular 2 quickstart code. We hope you like typescript:

```
import { Component } from '@angular/core';
@Component({
  selector: '.my-app',
  template: '<h1>NG2 from Exercise 3</h1>'
})
export class AppComponent { }
```

You can do whatever in this module however, please notice how we changed the selector to `.my-app`. In Angular 2, the selector can be anything. By changing it to a class name, it'll be easier for us to choose where we want to bootstrap our angular 2 component.

Register static resource directory

Next, let's register the static directory we just placed our script into. To register, you need to add ZCML registration for the static directory your script is in. Add this to the `exercise3/configure.zcml` file:

```
<plone:static
     directory="static"
     type="plone"
     name="exercise3"
     />
```

Build the file with webpack

Our deployment is built using webpack:

```
cd exercise3/static
webpack
```

Whenever you make a change to your component files, webpack will auto re-build the distribution.

Register JavaScript resource as a bundle

Register our script as a JavaScript resource with Plone. In the `exercise3/profiles/default/registry.xml` file, add configuration to register your script:

```
<records prefix="plone.bundles/exercise3"
          interface='Products.CMFPlone.interfaces.IBundleRegistry'>
  <value key="merge_with">default</value>
  <value key="enabled">True</value>
  <value key="compile">False</value>
  <value key="jscompilation">++plone++exercise3/exercise3-compiled.min.js</value>
  <value key="csscompilation">++plone++exercise3/exercise3-compiled.css</value>
  <value key="last_compilation">2016-10-04 00:00:00</value>
</records>
```

A couple notes about this configuration:

- `merge_with` tells plone to combine this file with the default Plone bundles
- `compile` is distinguish this bundle as one that is compiled outside of Plone
- `jscompilation` and `csscompilation` are what Plone uses as the final compiled output

Installation

1. Start up your Plone instance
2. Install the `Exercise 3` add-on

Warning: To make sure your resource registry configuration changes apply, you'll need to be in development mode. You can also toggle development mode on and off, click save, to force configuration to be re-built after changes instead of keeping development mode on.

Running

It's up to you how to apply the component class name to an element of your choice. A couple options available to you are:

1. use TinyMCE source view and add `class="my-app"` onto any tag
2. customize the theme on your site and add it to an element in your theme file or use a diazo rule diazo rule to dynamically add the class to an element

Development

To make sure your changes are loaded after every build with webpack, make sure to go into Site setup -> Resource registries and enabled development mode.

Production

Production for this is simple when you're no longer in development mode on your Plone site. Webpack rebuilds the JavaScript distribution on every change.

Exercise 4: NG2 APP in logged in bundle

Warning: This exercise requires a working buildout using a fork of the collective.jstraining package.

For this exercise, we will add an angular 2 application to a plone bundle.

We have most of the angular 2 boilerplate code created for you so let's just finish up a few things so you can customize it.

We will be working in the `exercise4` directory of the collective.jstraining package.

Bootstrap

Install npm dependencies:

```
cd exercise4/static
npm install
```

Add your angular 2 component

In your `exercise4/static/app` directory, add a file named `app.component.ts`. Use this file to do anything you would like to the page. This example will stick with the angular 2 quickstart code. We hope you like typescript:

```
import { Component } from '@angular/core';
@Component({
  selector: '.my-app',
  template: '<h1>NG2 from Exercise 4</h1>'
})
export class AppComponent { }
```

You can do whatever in this module however, please notice how we changed the selector to `.my-app`. In Angular 2, the selector can be anything. By changing it to a class name, it'll be easier for us to choose where we want to bootstrap our angular 2 component.

Register static resource directory

Next, let's register the static directory we just placed our script into. To register, you need to add ZCML registration for the static directory your script is in. Add this to the `exercise4/configure.zcml` file:

```
<plone:static
     directory="static"
     type="plone"
     name="exercise4"
     />
```

Build the file with webpack

Our deployment is built using webpack:

```
cd exercise4/static
webpack
```

Whenever you make a change to your component files, webpack will auto re-build the distribution

Register JavaScript resource

Let's register our script as a JavaScript resource with Plone. In the `exercise4/profiles/default/registry.xml` file, add configuration to register your script:

```
<records prefix="plone.bundles/exercise4"
          interface='Products.CMFPlone.interfaces.IBundleRegistry'>
  <value key="merge_with">logged-in</value>
  <value key="enabled">True</value>
  <value key="compile">False</value>
  <value key="expression">python: member is not None</value>
  <value key="jscompilation">++plone++exercise4/exercise4-compiled.min.js</value>
  <value key="csscompilation">++plone++exercise4/exercise4-compiled.css</value>
  <value key="last_compilation">2016-10-04 00:00:00</value>
</records>
```

Pay attention to this part of the exercise. Here we merge the bundle with `logged-in` instead of `default`. We also added an `expression` configuration option to specify that we only want this bundle to load for logged in users.

Installation

1. Start up your Plone instance
2. Install the `Exercise 4` add-on

Running

It's up to you how to apply the component class name to an element of your choice. A couple options available to you are:

1. use TinyMCE source view and add `class="my-app"` onto any tag

2. customize the theme on your site and add it to an element in your theme file or use a diazo rule diazo rule to dynamically add the class to an element

> **Warning:** To make sure your resource registry configuration changes apply, you'll need to be in development mode. You can also toggle development mode on and off, click save, to force configuration to be re-built after changes instead of keeping development mode on.

Development

To make sure your changes are loaded after every build with webpack, make sure to go into Site setup -> Resource registries and enabled development mode.

Production

Production for this is simple when you're no longer in development mode on your Plone site. Webpack rebuilds the JavaScript distribution on every change.

Exercise 5: Gallery integration with theme

> **Warning:** This exercise requires a working buildout using a fork of the collective.jstraining package.

In this exercise, we'll be walking through how to include custom JavaScript into your theme.

This example essentially re-uses the Barceloneta theme. It's more important to pay attention to how the integration with the theme works, than worrying about diazo/theming details for this exercise.

We will be working in the `exercise5` directory of the collective.jstraining package.

Add your JavaScript file

The lightGallery distribution files are already included in the `collective.jstraining` package you're working with.

In your `exercise5/theme` directory, add a file named `integration.js`. We'll use this file to integrate with Plone's album view:

```
require([
  'jquery',
  '++theme++exercise5/lightgallery/js/lightgallery.min'
], function($){
  $(document).ready(function() {
    var $photos = $('.photoAlbumEntry a');
    if($photos.size() > 0){
      // we're on an album view page
      // we need to adjust links so the work nicely with light gallery
      $photos.each(function(){
        var $a = $(this);
        $a.attr('href', $a.attr('href').replace('/view', ''));
      });
      $("#content-core").lightGallery({
        selector: '.photoAlbumEntry a'
      });
    }
  });
});
```

Let’s talk about each part of this file in detail...

Require the lightGallery JavaScript:

```
...
require([
  'jquery',
  '++theme++exercise5/lightgallery/js/lightgallery.min'
], function($){
...
```

This tells RequirejS to load the jQuery and the lightGallery JavaScript.

What is important to pay attention to in this example is that we’re seeing if there are any `photoAlbumEntry` elements on the page.

If there are any, we modify the DOM structure slightly to work seemlessly with lightGallery:

```
...
var $photos = $('.photoAlbumEntry a');
if($photos.size() > 0){
  // we're on an album view page
  // we need to adjust links so the work nicely with light gallery
  $photos.each(function(){
    var $a = $(this);
    $a.attr('href', $a.attr('href').replace('/view', ''));
  });
...
```

Finally, we call the lightGallery initialization with our configuration:

```
...
$("#content-core").lightGallery({
  selector: '.photoAlbumEntry a'
});
...
```

Including JavaScript/CSS into your theme

For JavaScript and CSS, you can include resources with convenience theme configuration settings of `development-css`, `production-css`, `development-js` and `production-js`.

Since we’re reusing the existing Barceloneta theme with this example though, we’ll simple include the JavaScript/CSS into the theme `index.html` file.

CSS

At the bottom of the head section in the `index.html` file, add:

```
<link rel="stylesheet" type="text/css"
        href="../++theme++exercise5/lightgallery/css/lightgallery.min.css" />
```

JavaScript

At the bottom of the `index.html` file, before the `</body>` closing tag, add:

```
<script src="../++theme++exercise5/integration.js"></script>
```

Installation

1. Start up your Plone instance
2. Install the `Exercise 5` add-on

Trying it out

1. Create a folder and add some images to it in your Plone site.
2. Specify `Album view` for your folder.
3. Now when you click on an image, it should show the gallery viewer.

Production

In this example, there is no difference with development vs production.

You can combine this example with other examples of building JavaScript projects to build, compile and minify your resources.

Exercise 6: Simple Pattern

Warning: This exercise requires a working buildout using a fork of the collective.jstraining package.

In this exercise, we'll be walking through creating a simple Plone pattern.

We will be working in the `exercise6` directory of the collective.jstraining package.

Add your pattern file

First off, in your `exercise6/static` directory, add a file named `pattern.js`. Use this file to build your pattern. This example will stay very simple we'll use jQuery to do a simply modify the content of an element:

```
define([
  'jquery',
  'mockup-patterns-base',
], function($, Base) {
  'use strict';
  var Pattern = Base.extend({
    name: 'exercise6',
    trigger: '.pat-exercise6',
    parser: 'mockup',
    defaults: {
    },
    init: function() {
      var that = this;
      that.$el.append(' <span>Exercise 6 was here</span>');
    }
  });

  return Pattern;
});
```

For more details on how to write a mockup pattern, utilize the various resources available.

- Minimal pattern
- Mockup docs
- Patternslib

In our example, we're using the RequireJS `define` function to define our pattern as a JavaScript module.

Integrating with LESS

Add a `pattern.less` file to the `exercise6/static` directory and provide whatever styles you'd like for your pattern:

```
.pat-exercise6 {
  color: red;
}
```

Creating your bundle

To register the pattern, we'll create a bundle. Recall the difference between using `require` and `define` from the RequireJS docs.

Our bundle will use the `require` function to include the JavaScript module pattern we created our previously.

Create a `bundle.js` file in your `exercise6/static` directory:

```
require([
  'exercise6'
], function() {
  'use strict';
});
```

The only thing we're doing in this file is including the `exercise6` module we defined earlier–that's it. Bundles can do more as well. Then can include initialization code for example. See Plone's default bundle.

Bundles more or less tell the compiler what we care about loading. They do the dependency resolution and include the modules that were required with them.

Register static resource directory

Next, let's register the static directory we just placed our script into. To register, you need to add ZCML registration for the static directory your script is in. Add this to the `exercise6/configure.zcml` file:

```
<plone:static
     directory="static"
     type="plone"
     name="exercise6"
     />
```

Register your bundle

Registering your bundle is done by adding Generic Setup xml configuration to the Plone registry. This is done in the `registry.xml` file in the `profiles/default` directory.

Resource

Resource is done exactly the same as in Exercise 1:

```
<records prefix="plone.resources/exercise6"
          interface='Products.CMFPlone.interfaces.IResourceRegistry'>
  <value key="js">++plone++exercise6/pattern.js</value>
</records>
```

Bundle resource

The bundle resource is just another resource registration like any other. Remember, the only difference here is in the content of the JavaScript file. One file uses `require`, the other uses `define`. Addditionally, we include our CSS/LESS dependencies here:

```
<records prefix="plone.resources/bundle-exercise6"
          interface='Products.CMFPlone.interfaces.IResourceRegistry'>
  <value key="js">++plone++exercise6/bundle.js</value>
  <value key="css">
    <element>++plone++exercise6/pattern.less</element>
  </value>
</records>
```

Bundle

Finally, let's create our bundle registration:

```
<records prefix="plone.bundles/exercise6"
          interface='Products.CMFPlone.interfaces.IBundleRegistry'>
  <value key="resources">
    <!-- reference to bundle resource definition -->
    <element>bundle-exercise6</element>
  </value>
  <value key="merge_with">default</value>
  <value key="enabled">True</value>
  <value key="jscompilation">++plone++exercise6/exercise6-compiled.min.js</value>
  <value key="csscompilation">++plone++exercise6/exercise6-compiled.css</value>
  <value key="last_compilation">2016-10-04 00:00:00</value>

  <!-- so we don't include these modules multiple times -->
  <value key="stub_js_modules">
    <element>jquery</element>
    <element>mockup-patterns-base</element>
  </value>
</records>
```

Installation

1. Start up your Plone instance
2. Install the `Exercise 6` add-on

Running

At this point, we have no compiled version of the code that we're running with so our code doesn't do anything.

1. Go into `Site Setup` -> `Resource Registries`
2. Check "Development Mode"
3. Select to develop JavaScript and CSS for the `exercise6` bundle
4. Click save

This should load your JavaScript and LESS files now; however, we don't have any elements with the `pat-exercise6` class assigned to them.

It's up to you how to apply the pattern class to an element of your choice. A couple options available to you are:

1. use TinyMCE source view and add `class="pat-exercise6"` onto any `p` tag
2. customize the theme on your site and add it to an element in your theme file or use a diazo rule diazo rule to dynamically add the class to an element

Production

To build our bundle, we'll utilize the `plone-compile-resources` script that Plone ships with.

Warning: If you're not running a ZEO setup, you'll need to shut down your Plone instance since the ZODB in this mode does not allow multiple processes to access it at the same time.

An example command will look like this:

```
./bin/plone-compile-resources --site-id=Plone --bundle=exercise6
```

Once this command finishes, your bundle is built and will be deployed with your package package.

Exercise 7: Using a pattern in a z3c form widget

Warning: This exercise requires a working buildout using a fork of the collective.jstraining package.

This exercise will go through adding a widget that checks the minimum size of an image before it is uploaded.

We will be working in the `exercise7` directory of the collective.jstraining package.

Add your JavaScript file

First off, in your `exercise7/static` directory, add a file named `script.js`. Use this file to do anything you would like to the page:

```
/* global require, FileReader, Image */

require([
  'jquery',
  'mockup-patterns-base',
], function($, Base) {
  'use strict';

  Base.extend({
    name: 'exercise7',
    trigger: '.pat-exercise7',
    parser: 'mockup',
    defaults: {
      minHeight: 200,
      minWidth: 200
    },
    init: function() {
      var that = this;

      that.$el.on('change', function(){
        if(this.files.length === 0){
          return;
        }

        var fr = new FileReader();
        fr.onload = function() {
          var img = new Image();
          img.onload = function() {
            if(img.width < that.options.minWidth ||
               img.height < that.options.minHeight){
              alert('Invalid image size. The image must be at least ' +
                    that.options.minWidth + 'x' + that.options.minHeight + '.');
              that.$el[0].value = '';
            }
          };
          img.src = fr.result;
        };
        fr.readAsDataURL(this.files[0]);
      });
    }
  });
});
```

This pattern simply has `minWidth` and `minHeight` options and when a file is selected for upload, will check to make sure it is a valid size.

Register static resource directory

Next, let's register the static directory we just placed our script into. To register, you need to add ZCML registration for the static directory your script is in. Add this to the `exercise7/configure.zcml` file:

```
<plone:static
     directory="static"
     type="plone"
     name="exercise7"
     />
```

Register JavaScript resource

Register our script as a JavaScript resource with Plone. In the `exercise7/profiles/default/registry.xml` file, add configuration to register your script:

```
<records prefix="plone.resources/exercise7"
         interface='Products.CMFPlone.interfaces.IResourceRegistry'>
    <value key="js">++plone++exercise7/script.js</value>
</records>
```

Create a custom widget

Our custom widget will apply to all lead images. Add a file `widget.py` to your `exercise7` directory with the follow contents:

```
from .interfaces import IExercise7Layer
from .interfaces import IMinSizeImageWidget
from plone.app.contenttypes.behaviors.leadimage import ILeadImage
from plone.formwidget.namedfile.widget import NamedImageWidget
from Products.CMFPlone.resources import add_resource_on_request
from z3c.form.interfaces import IFieldWidget
from z3c.form.util import getSpecification
from zope.component import adapter
from zope.interface import implementer
from zope.interface import implements

import json
import z3c.form.widget


class MinSizeImageWidget(NamedImageWidget):
    """A widget for a named file object
    """
    implements(IMinSizeImageWidget)
    def pattern_options(self):
        # provide the pattern options
        return json.dumps({
            'minHeight': 300,
            'minWidth': 300
        })

    def render(self):
        # add the registered resource
        add_resource_on_request(self.request, 'exercise7')
        return super(MinSizeImageWidget, self).render()


@adapter(getSpecification(ILeadImage['image']), IExercise7Layer)
@implementer(IFieldWidget)
def LeadImageMinSizeImageFieldWidget(field, request):
    widget = z3c.form.widget.FieldWidget(field, MinSizeImageWidget(request))
    return widget
```

Notice in the `render` method we utilize the `add_resource_on_request` function to load our pattern.

The code for `image_widget.pt` is already provided for this example since it is quite long. Review the file and notice where we are passing the value from the `pattern_options` method into our widget.

Register widget customization

Next, we need to register our custom widget so it is used. In your `configure.zcml` file, add the following:

```
<adapter factory=".widget.LeadImageMinSizeImageFieldWidget" />
<z3c:widgetTemplate
  mode="input"
  widget=".interfaces.IMinSizeImageWidget"
  layer=".interfaces.IExercise7Layer"
  template="image_widget.pt"
  />
```

Installation

1. Start up your Plone instance
2. Install the `Exercise 7` add-on

Now, try to add/edit a lead image to content on the site.

> **Warning:** To make sure your resource registry configuration changes apply, you'll need to be in development mode. You can also toggle development mode on and off, click save, to force configuration to be re-built after changes instead of keeping development mode on.

Exercise 8: Pattern wrapping a 3rd party library

Warning: This exercise requires a working buildout using a fork of the collective.jstraining package.

In this exercise, we'll be walking through wrapping the tablesorter JavaScript library into a pattern

We will be working in the `exercise8` directory of the collective.jstraining package.

Add your pattern file

First off, in your `exercise8/static` directory, add a file named `pattern.js`. Use this file to build your pattern. This example will simply load and initialize the table sorter js:

```
/* global require */

require([
  'jquery',
  'mockup-patterns-base',
  'tablesorter'
], function($, Base) {
  'use strict';

  /* combining bundle and pattern in same file this example */

  Base.extend({
    name: 'tablesorter',
    trigger: '.pat-tablesorter',
    parser: 'mockup',
    defaults: {
    },
    init: function() {
      var that = this;
      that.$el.tablesorter();
    }
  });

});
```

Notice in this example how we're not using `define` for this pattern. In this example, we are defining our pattern right inside what will be our bundle.

`tablesorter` will be our registered 3rd party library include.

Register static resource directory

Register the static directory we just placed our script into. To register, you need to add ZCML registration for the static directory your script is in. Add this to the `exercise8/configure.zcml` file:

```
<plone:static
     directory="static"
     type="plone"
     name="exercise8"
     />
```

Register your bundle

Registering your bundle is done by adding Generic Setup xml configuration to the Plone registry. This is done in the `registry.xml` file in the `profiles/default` directory.

Tablesorter

Resource is done exactly the same as in Exercise 1:

```
<records prefix="plone.resources/tablesorter"
         interface='Products.CMFPlone.interfaces.IResourceRegistry'>
  <value key="js">++plone++exercise8/jquery.tablesorter.min.js</value>
</records>
```

Bundle resource

Our pattern is a bundle-able resource since it uses the `require` function instead of the `define` function:

```
<records prefix="plone.resources/exercise8"
            interface='Products.CMFPlone.interfaces.IResourceRegistry'>
    <value key="js">++plone++exercise8/pattern.js</value>
    <value key="css">
      <element>++plone++exercise8/pattern.less</element>
    </value>
  </records>
```

Bundle

Finally, let’s create our bundle registration:

```
<records prefix="plone.bundles/exercise8"
          interface='Products.CMFPlone.interfaces.IBundleRegistry'>
  <value key="resources">
    <element>exercise8</element>
  </value>
  <value key="merge_with">default</value>
  <value key="enabled">True</value>
  <value key="jscompilation">++plone++exercise8/exercise8-compiled.min.js</value>
  <value key="csscompilation">++plone++exercise8/exercise8-compiled.css</value>
  <value key="last_compilation">2016-10-04 00:00:00</value>
  <value key="stub_js_modules">
    <element>jquery</element>
    <element>mockup-patterns-base</element>
  </value>
</records>
```

Installation

At this point, we have all the files necessary to run the pattern.

1. Start up your Plone instance

2. Install the `Exercise 8` add-on

Running

At this point, we have no compiled version of the code that we're running with so our code doesn't do anything.

1. Go into `Site Setup` -> `Resource Registries`
2. Check "Development Mode"
3. Select to develop JavaScript and CSS for the `exercise8` bundle
4. Click save

This should load your JavaScript and LESS files now; however, we don't have any elements with the `pat-exercise8` class assigned to them.

It's up to you how to apply the pattern class to an element of your choice. A couple options available to you are:

1. use TinyMCE source view and add `class="pat-tablesorter"` onto any `table` tag. You need to use `th` tags for the top row in your header in order for tablesorter to know to do anything.
2. customize the theme on your site and add it to an element in your theme file or use a diazo rule diazo rule to dynamically add the class to an element

Production

To build our bundle, we'll utilize the `plone-compile-resources` script that Plone ships with.

> **Warning:** If you're not running a ZEO setup, you'll need to shut down your Plone instance since the ZODB in this mode does not allow multiple processes to access it at the same time.

An example command will look like this:

```
./bin/plone-compile-resources --site-id=Plone --bundle=exercise8
```

Once this command finishes, your bundle is built and will be deployed with your package package.

Exercise 9: Pattern with react

> **Warning:** This exercise requires a working buildout using a fork of the collective.jstraining package.

In this exercise, we'll be walking through creating a pattern that uses ReactJS.

We will be working in the `exercise9` directory of the collective.jstraining package.

Add your pattern file

First off, in your `exercise9/static` directory, add a file named `pattern.js`. Use this file to build your pattern. This example will bind a React component to a pattern element:

```
/* global require */

require([
  'jquery',
  'mockup-patterns-base',
  'exercise9-react'
], function($, Base, R) {
  'use strict';
  /* combining bundle and pattern in same file this example */

  var D = R.DOM;


  var Exercise9Component = R.createClass({
    render: function(){
      return D.div({}, [
        D.span({}, 'Foobar rendered by exercise 9')
      ]);
    }
  });

  Base.extend({
    name: 'exercise9',
    trigger: '.pat-exercise9',
    parser: 'mockup',
    defaults: {
    },
    init: function() {
      var that = this;
      R.render(R.createElement(Exercise9Component, that.options), that.$el[0]);
    }
  });

});
```

Notice that the `init` of the pattern utilizes the React element binding syntax. From there, react takes over and options from the pattern go into `props` for the React component.

Register static resource directory

Next, let's register the static directory we just placed our script into. To register, you need to add ZCML registration for the static directory your script is in. Add this to the `exercise9/configure.zcml` file:

```
<plone:static
     directory="static"
     type="plone"
     name="exercise9"
     />
```

Register your bundle

Registration is done exactly like the other examples:

```
<records prefix="plone.resources/exercise9-react"
         interface='Products.CMFPlone.interfaces.IResourceRegistry'>
  <value key="js">++plone++exercise9/react.min.js</value>
  <value key="css">
  </value>
</records>

<records prefix="plone.resources/exercise9"
          interface='Products.CMFPlone.interfaces.IResourceRegistry'>
  <value key="js">++plone++exercise9/pattern.js</value>
  <value key="css">
    <element>++plone++exercise9/pattern.less</element>
  </value>
</records>

<records prefix="plone.bundles/exercise9"
          interface='Products.CMFPlone.interfaces.IBundleRegistry'>
  <value key="resources">
    <element>exercise9</element>
  </value>
  <value key="merge_with">default</value>
  <value key="enabled">True</value>
  <value key="jscompilation">++plone++exercise9/exercise9-compiled.min.js</value>
  <value key="csscompilation">++plone++exercise9/exercise9-compiled.css</value>
  <value key="last_compilation">2016-10-04 00:00:00</value>
  <value key="stub_js_modules">
    <element>jquery</element>
    <element>mockup-patterns-base</element>
  </value>
</records>
```

Installation

At this point, we have all the files necessary to run the pattern.

1. Start up your Plone instance
2. Install the `Exercise 9` add-on

Running

At this point, we have no compiled version of the code that we're running with so our code doesn't do anything.

1. Go into `Site Setup` -> `Resource Registries`
2. Check "Development Mode"
3. Select to develop JavaScript and CSS for the `exercise9` bundle
4. Click save

This should load your JavaScript and LESS files now; however, we don't have any elements with the `pat-exercise9` class assigned to them.

It's up to you how to apply the pattern class to an element of your choice. A couple options available to you are:

1. use TinyMCE source view and add `class="pat-exercise9"` onto any tag

2. customize the theme on your site and add it to an element in your theme file or use a diazo rule diazo rule to dynamically add the class to an element

Production

To build our bundle, we'll utilize the `plone-compile-resources` script that Plone ships with.

> **Warning:** If you're not running a ZEO setup, you'll need to shut down your Plone instance since the ZODB in this mode does not allow multiple processes to access it at the same time.

An example command will look like this:

```
./bin/plone-compile-resources --site-id=Plone --bundle=exercise9
```

Once this command finishes, your bundle is built and will be deployed with your package package.

Exercise 10: Customizing pattern

> **Warning:** This exercise requires a working buildout using a fork of the collective.jstraining package.

In this exercise, we'll be walking through customizing the livesearch pattern.

We will be working in the `exercise10` directory of the collective.jstraining package.

Add your pattern file

In your `exercise10/static` directory, add a file named `pattern.js`. Use this file to build your pattern. This example will define a new pattern to overwrite the existing livesearch pattern:

```
/* global require */

require([
  'jquery',
  'mockup-patterns-livesearch',
  'pat-registry'
], function($, Livesearch, registry) {
  'use strict';
  /* combining bundle and pattern in same file this example */


  // first, unregister existing pattern
  delete registry.patterns.livesearch;
  delete $.fn.patLivesearch;


  // creating new pattern automatically registers it
  Livesearch.extend({
    name: 'livesearch',
    trigger: '.pat-livesearch',
    parser: 'mockup',
    init: function() {
      var that = this;
      Livesearch.prototype.init.call(that);

      // all we're doing in this customization is defaulting to searching
      // current section
      $('.searchSection input', that.$el)[0].checked = true;
    }
  });

});
```

Pay close attention to what we're doing here:

```
...
delete registry.patterns.livesearch;
delete $.fn.patLivesearch;
...
```

We're deleting the existing registration of the livesearch pattern.

Next, we're extending the existing pattern:

```
...
Livesearch.extend({
...
```

And just overriding the `init` function to provide our customization(default search current section):

```
...
$('.searchSection input', that.$el)[0].checked = true;
...
```

Register static resource directory

Next, let's register the static directory we just placed our script into. To register, you need to add ZCML registration for the static directory your script is in. Add this to the `exercise10/configure.zcml` file:

```
<plone:static
     directory="static"
     type="plone"
     name="exercise10"
     />
```

Register your bundle

Again, registration is done examctly the same as previous exercises:

```
<?xml version="1.0"?>
<registry>

  <records prefix="plone.resources/exercise10"
            interface='Products.CMFPlone.interfaces.IResourceRegistry'>
    <value key="js">++plone++exercise10/pattern.js</value>
    <value key="css">
```

```
      <element>++plone++exercise10/pattern.less</element>
    </value>
  </records>

  <records prefix="plone.bundles/exercise10"
            interface='Products.CMFPlone.interfaces.IBundleRegistry'>
    <value key="resources">
      <element>exercise10</element>
    </value>
    <value key="merge_with">default</value>
    <value key="enabled">True</value>
    <value key="jscompilation">++plone++exercise10/exercise10-compiled.min.js</value>
    <value key="csscompilation">++plone++exercise10/exercise10-compiled.css</value>
    <value key="last_compilation">2016-10-04 00:00:00</value>
    <value key="stub_js_modules">
      <element>jquery</element>
      <element>mockup-patterns-livesearch</element>
      <element>pat-registry</element>
    </value>
  </records>

</registry>
```

Installation

We have all the files necessary to run the pattern now.

1. Start up your Plone instance
2. Install the `Exercise 10` add-on

Running

At this point, we have no compiled version of the code that we're running with so our code doesn't do anything.

1. Go into `Site Setup` -> `Resource Registries`
2. Check "Development Mode"
3. Select to develop JavaScript and CSS for the `exercise10` bundle
4. Click save

Now, you should see the livesearch pattern default to searching the current section.

Production

To build our bundle, we'll utilize the `plone-compile-resources` script that Plone ships with.

> **Warning:** If you're not running a ZEO setup, you'll need to shut down your Plone instance since the ZODB in this mode does not allow multiple processes to access it at the same time.

An example command will look like this:

```
./bin/plone-compile-resources --site-id=Plone --bundle=exercise10
```

Once this command finishes, your bundle is built and will be deployed with your package package.

Automating Plone Deployment

Note: This training is meant to be used in a course or read and worked through by an individual user. Instructors should note that this makes it more discursive than it would be if it was only meant for classroom use. Many sections may be zipped through in a class, noting to students that the full text is available for review.

Contents:

Introduction

The subject of this training is the deployment of Plone for production purposes. We will, in particular, be focusing on automating deployment using tools which can target a fresh Linux server and create on it an efficient, robust server.

That target server may be a cloud server newly created on AWS, Linode or DigitalOcean. Or, it may be a virtual machine created for testing on your own desk or laptop.

Our goal is that these deployments be *repeatable*. If we run the automated deployment multiple times against multiple cloud servers, we should get the same results. If we run the automated deployment against a virtual machine on our laptop, we should be able to test it as if it was a matching cloud server.

The tools we use for this purpose reflect the opinions of the Plone Installer Team. *We are opinionated*. With a great many years of experience administering servers and Plone, we feel we have a right to our opinions. But, most importantly, we know we have to make choices and support those choices.

The tools we use may not be the ones you would choose.

They may not be the ones we would choose this month if we were starting over.

But, they are tools widely used in the Plone community. They are well-understood, and you should get few “I’ve never heard of that” complaints if you ask questions of the Plone community.

Our big choices

Linux

> BSD is great. OS X is familiar. Windows works just fine. But our experience and the majority experience in the Plone community is with Linux for production servers. That doesn’t mean you have to use Linux for your laptop or desktop; anything that runs Python is likely fine.

Major distributions

> We’re supporting two target distribution families: Debian and EL (RedHat/CentOS). We’re going to try to keep this working on the most recent LTS (Long-Term Support release) or its equivalent.

Platform packages

> We use platform packages whenever possible. We want the non-Plone components on your server to be automatically updatable using your platform tools. If a platform package is usable, we’ll use it even if it isn’t the newest, coolest version.

Ansible

> There are all sorts of great tools for automating deployment. People we respect have chosen Puppet, Salt/Minion and lots of other tools. We chose Ansible because it requires no preinstalled server component, it's written in Python, and its configuration language is YAML, which is awfully easy to read.

And ...

> We'll discuss particular parts of the deployment stack in the next section.

Intro to Plone Stack

If you haven't read the first couple of chapters of Guide to deploying and installing Plone in production, take a moment to do so. You'll want to be familiar with the main components of a typical Plone install for deployment and know when each is vital and when unnecessary.

The Plone Ansible Playbook makes choices for each generic component.

You are not stuck with our choices. If, for example, you wish to use Apache rather than Nginx for the web server component, that won't be a particular problem. You'll just need to do more work to customize.

Intro to Ansible

Ansible is an open-source configuration management, provisioning and application deployment platform written in Python and using YAML (YAML Ain't Markup Language) as a configuration language. Ansible makes its connections from your computer to the target machine using SSH.

There is no server-side component other than an SSH server. General familiarity with SSH is very desirable if you're using Ansible – as well as being a baseline skill for server administration.

Installation

Ansible is usually installed on the orchestrating computer – typically your desktop or laptop. It is a large Python application (though a fraction the size of Plone!) that needs many specific Python packages from the Python Package Index (PyPI).

That makes Ansible a strong candidate for a Python **virtualenv** installation If you don't have **virtualenv** installed on your computer, do it now.

virtualenv may be installed via an OS package manager, or on a Linux or BSD machine with the command:

```
sudo easy_install-2.7 virtualenv
```

Once you've got **virtualenv**, use it to create a working directory containing a virtual Python:

```
virtualenv ansible_work
```

Then, install Ansible there:

```
cd ansible_work
bin/pip install ansible
```

Now, to use Ansible, activate that Python environment.

```
source bin/activate
ansible
```

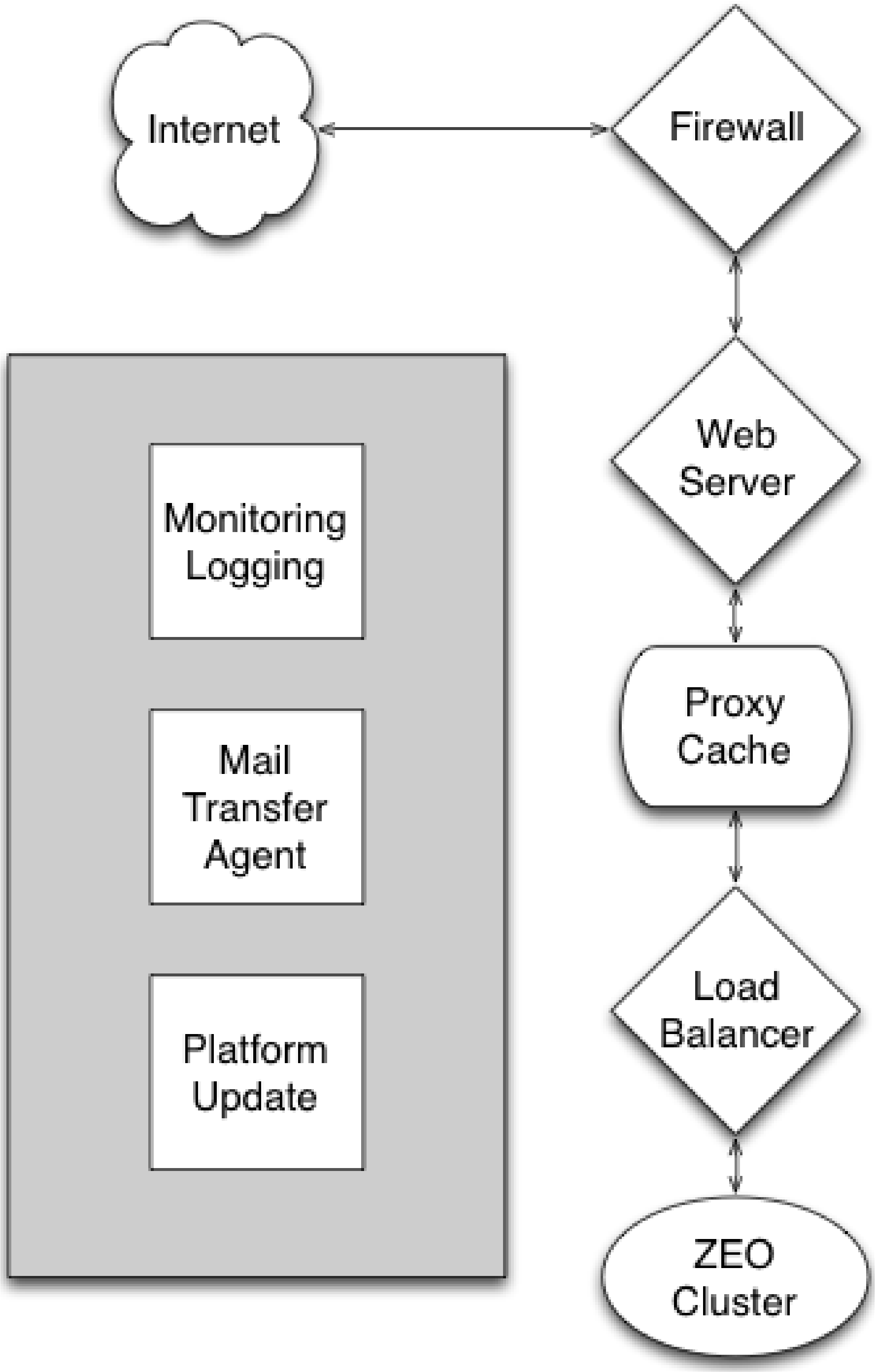


Fig. 2.1: The generic components of a full-stack Plone installation. Not all are always used.

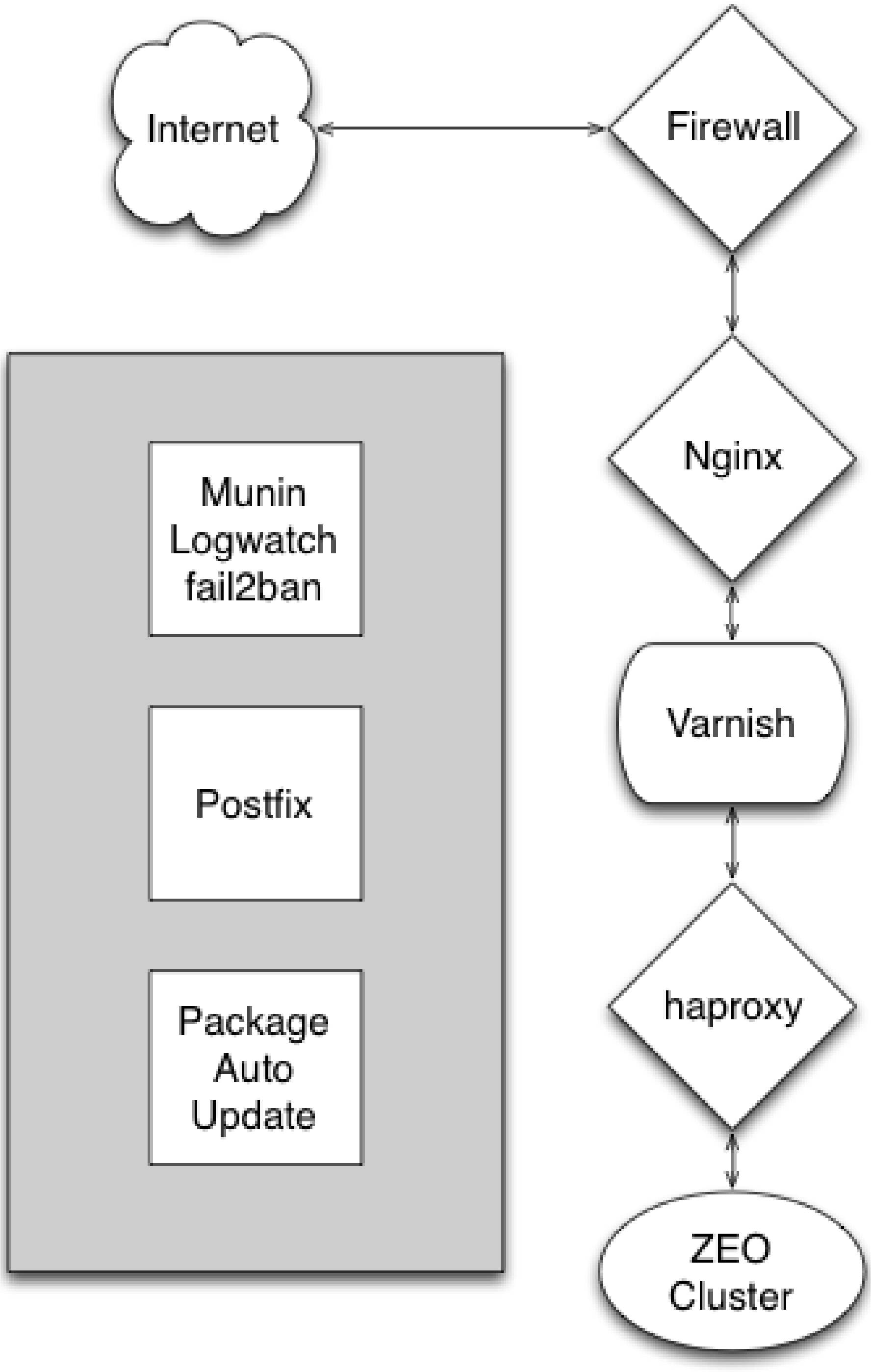


Fig. 2.2: The specific components used in Plone’s Ansible Playbook.

Note: Trainers: check to make sure everyone understands the basic `source activate` mechanism.

Now, let's get a copy of the *Plone Ansible Playbook*. Make sure you're logged in to your `ansible_work` directory.

Unless you're participating in the development of the playbook, or need a particular fix, you'll want to check out the `STABLE` branch. The `STABLE` branch is a pointer to the last release of the playbook.

```
git clone -b STABLE --single-branch https://github.com/plone/ansible-playbook.git
```

Or,

```
git clone https://github.com/plone/ansible-playbook.git
cd ansible-playbook
git checkout STABLE
```

That gives you the Plone Ansible Playbook. You'll also need to install a few Ansible roles. Roles are Ansible playbooks packaged for distribution. Fortunately, you may pick up everything with a single command.

```
cd ansible-playbook
ansible-galaxy install -p roles -r requirements.yml
```

If you forget that command, it's in the short README.rst file in the playbook.

Note: The rationale for checking the Plone Ansible Playbook out inside the virtualenv directory is that it ties the two together. Months from now, you'll know that you can use the playbook with the Python and Ansible packages in the virtualenv directory. We check out the playbook as a subdirectory of the virtualenv directory so that we can search our playbooks and roles without having to search the whole virtualenv set of packages.

Ansible basics

Connecting to remote machines

To use Ansible to provision a remote server, we have two requirements:

1. We must be able to connect to the remote machine using **`ssh`**; and,
2. We must be able to issue commands on the remote server as `root` (superuser), usually via **`sudo`**.

You'll need to familiarize yourself with how to fulfill these requirements on the cloud/virtual environment of your choice. Examples:

Using Vagrant/virtualbox

> You will initially be able to log in as the "vagrant" user using a private key that's in a file created by Vagrant. The user "vagrant" may issue **`sudo`** commands with no additional password.

Using Linode

> You'll set a root password when you create your new machine. If you're willing to use the root user directly, you will not need a **`sudo`** password.

When setting up a Digital Ocean machine

> New machines are typically created with a root account that contains your ssh public key as an authorized key.

AWS

> AWS EC2 instances are typically created with a an account named "root" or a short name for the OS, like "ubuntu", that contains your ssh public key as an authorized key. Passwordless **sudo** is pre-enabled for that account.

The most important thing is that you know your setup. Test that knowledge by trying an ssh login and issuing a superuser command.

```
ssh myuser@myhost.com   # (what user/hostname did you use? are you asked a password?)
...
myhost.com $ sudo ls  # (are you asked for your password?)
```

Inventories

Ansible is usually run on a local computer, and it usually acts on one or more remote machines. We tell Ansible how to connect to remote machines by maintaining a text inventory file.

There is a sample inventory configuration file in your distribution. It's meant for use with a Vagrant-style virtualbox.

```
cat vbox.cfg
```

```
myhost ansible_port=2222 ansible_host=127.0.0.1 ansible_user=vagrant ansible_private_
↪key_file=~/.vagrant.d/insecure_private_key
```

This inventory file is complicated by the fact that a virtualbox typically has no DNS host name and uses a non-standard port and a special SSH key file. So, we have to specify all those things.

If we were using a DNS-known hostname and our standard ssh key files, it could be much simpler:

```
direct.newhost.com ansible_ssh_user=root
```

Ansible inventory files may list multiple hosts and may have aliases for groups of hosts. See https://docs.ansible.com for details.

Playbooks

We're going to cover just enough on Ansible playbooks to allow you to read and customize Plone's playbook. Ansible's documentation is excellent if you want to learn more.

In Ansible, an individual instruction for the setup of the remote server is called a _task_. Here's a task that makes sure a directory exists.

This uses the Ansible `file` module to check to see if a directory exists with the designated mode. If it doesn't, it's created.

Tasks may also have execution conditions expressed in Python syntax and may iterate over simple data structures.

In addition to tasks, Ansible's basic units are *host* and *variable* specifications.

An Ansible *playbook* is a specification of tasks that are executed for specified hosts and variables. All of these specifications are in YAML.

Quick intro to YAML

YAML isn't a markup language, and it isn't a programming language either. It's a data-specification notation. Just like JSON. Except that YAML – very much unlike JSON – is meant to be written and read by humans. The creators of YAML call it a "human friendly data serialization standard".

Note: YAML is actually a superset of JSON. Every JSON file is also a valid YAML file. But if we just fed JSON to the YAML parser, we'd be missing the point of YAML, which is human readability.

Basic types available in YAML include strings, booleans, floating-point numbers, integers, dates, times and date-times. Structured types are sequences (lists) and mappings (dictionaries).

Sequences are indicated by list-member lines with leading dashes:

```
- item one
- item two
- item three
```

Mappings are indicated with key/value pairs with colons separating keys and values:

```
one: item one
two: item two
three: item three
```

Complex data structures are designated with indentation:

```
# a mapping of sequences
american:
  - Boston Red Sox
  - Detroit Tigers
  - New York Yankees
national:
  - New York Mets
  - Chicago Cubs
  - Atlanta Braves

# a sequence of mappings
-
  name: Mark McGwire
  hr:   65
  avg:  0.278
-
  name: Sammy Sosa
  hr:   63
  avg:  0.288
```

Basic types read as you'd expect:

```
- one  # string "one"
- 1    # integer 1
- 1.0  # float 1.0
- True # boolean True
- true # also boolean True
- yes  # also boolean True
```

Finally, remember that this is a superset of JSON:

```
- {a: one, b: two}   # mapping
- [one, two, three]  # sequence
```

Want to turn YAML into Python data structures? Or Python into YAML? Python has several YAML parser/generators. The most commonly used is PyYAML.

Quick code to read YAML from the standard input and turn it into pretty-printed Python data:

```
#! /usr/bin/python

import yaml
import pprint
import sys

pprint.pprint(yaml.load(sys.stdin.read()), indent=2)
```

Quick intro to Jinja2

YAML doesn't have any built-in way to read a variable. Ansible uses the Jinja2 templating language for this purpose.

A quick example: Let's say we have a variable `timezone` containing the target server's desired timezone setting. We can use that variable in a task via Jinja2's double-brace notation: `{{ timezone }}`.

Jinja2 also supports limited Python expression syntax and can read object properties or mapping key/vaues with a dot notation:

```
{{ instance_config.plone_version < '5.0' }}
```

There are also various filters and tests available via a pipe notation. For example, we use the `default` filter to supply a default value if a variable is undefined.

```
- name: Set timezone variables
  tags: timezone
  copy: content={{ timezone|default("UTC\n") }}
        dest=/etc/timezone
        owner=root
        group=root
        mode=0644
        backup=yes
```

Jinja2 also is used as a full templating language whenever we need to treat a text file as a template to fill in variable values or execute loops or branching logic. Here's an example from the template used to construct a buildout.cfg:

```
zcml =
{% if instance_config.plone_zcml_slugs %}
{% for slug in instance_config.plone_zcml_slugs %}
    {{ slug }}
{% endfor %}
{% endif %}
```

Playbook structure

An Ansible "play" is a mapping (or dictionary) with keys for hosts, variables and tasks. A playbook is a sequence of such dictionaries.

A simple playbook:

```
- hosts: all
  vars:
    ... a dictionary of variables
  tasks:
    ... a sequence of tasks
```

The value of hosts could be a single host name, the name of a group of hosts, or “all”.

Variables

Notifications and handlers

We may also specify “handlers” that are run if needed.

```
- hosts: all
  vars:
    ... a dictionary of variables
  tasks:
    - name: Change webserver setup
      ...
      notify: restart webserver
    ...
  handlers:
    - name: restart webserver
      service: webserver
      state: restarted
```

Handlers are run if a matching notification is registered. A particular handler is only run once, even if several notifications for it are registered.

Roles

Ansible has various ways to include the contents of YAML files into your playbook. “Roles” do it in a more structured way – much more like a package. Roles contain their own variables, tasks and handlers. They inherit the global variable environment and you may pass particular variables when they are called.

Plone’s Ansible Playbook includes several roles for chores such as setting up the load balancer and web server. Other roles are fetched (the role source itself is fetched) by `ansible-galaxy` when we use it to set up requirements. Most are fetched from github.

An simple Ansible playbook using roles:

```
- hosts: all
  vars:
    ... a dictionary of variables
  pre-tasks:
    ... tasks executed before roles are used.
  roles:
    ... a sequence of role invocation mappings like:
    - role: haproxy
      var1: value1
      var2: value2
      when: install_loadbalancer|default(True)
    ...
  tasks:
    ... other tasks, executed after the roles
  handlers:
    ... handlers for our own tasks; roles usually have their own
```

If we want to pass variables to roles, we just add their keys and values to the mapping.

Take a look at the `when: install_loadbalancer|default(True)` line above. A `when` key in a role or task mapping sets a condition for execution. For conditionals like `when`, Ansible expects a Jinja2 expression.

We could also have expressed that `when` condition as `"{{ install_loadbalancer|default(True) }}"`. Ansible interprets all literal strings as little Jinja2 templates.

The Plone Playbook

Currently supported platforms

We currently support two Linux families: Debian and RHEL. *Support* means that the playbook knows how to load platform package dependencies and how to set up users, groups, and the platform's method for setting up daemons to start and stop with the operating system.

Note: There's no particular reason why we can't extend that support to other families, like BSD. All we need is a champion to take responsibility for extending and testing on other platforms.

Debian

> Our goal is to support the current Ubuntu LTS and the Debian equivalent. Currently we're doing a bit better than that. On Ubuntu we're supporting everything from Trusty to Xenial. On Debian, we're working with both Jessie and Wheezy.

RHEL

> We're currently only testing on CentOS 7. If you're using Plone on RHEL, we could use your help on extending that support.

Quick review of contents

Let's quickly review what you're getting when you check out the Plone Ansible Playbook.

Playbooks

We include two playbooks:

playbook.yml

> The main playbook that sets everything except the firewall.

firewall.yml

> A separate playbook to set up the software firewall. Most sysadmins have their own firewall experience, and may or may not choose to use this playbook.

roles

Roles are basically pre-packaged subroutines with their own default variables. Several roles are part of the Plone Ansible Playbook kit and will be present in your initial checkout. These include roles that set up the haproxy load balancer, varnish cache, nginx http server, postfix SMTP agent, munin-node monitoring, logwatch log analysis, message-of-the-day and a fancy setup for restarting ZEO clients.

Other roles, including the role that actually sets up Plone, are loaded when you use `ansible-galaxy` to fetch the items listed in `requirements.yml`. Except for the Plone server role, these are generally very generic Ansible Galaxy roles that we liked.

Vagrant

Vagrant/Virtualbox is a very handy way to test your playbook, both during development and for future maintenance. We include a couple of files to help you get started with Vagrant testing.

Vagrantfile

> A Vagrant setup file that will allow you to create guest virtual hosts for any of the platforms we support and will run Ansible as the provisioner with playbook.yml. This currently defaults to building a Trusty box, but you may pick others by naming them on the **`vagrant up`** command line.

vbox_host.cfg

> When you use vagrant commands, vagrant controls the ssh connection. `vbox_host.cfg` is an Ansible inventory file that should allow you to run your playbook directly (without the **`vagrant`** command) against your guest box.

Sample configurations

The playbook kit contains several sample configuration files.

sample-very-small.yml

> Targets a server with 512MB of memory and one CPU core. Sets up one ZEO client with two threads with very small object caches. No load balancer. Varnish cache is file-based.

sample-small.yml

> Targets a server with 1GB of memory and one CPU core. Sets up one ZEO client with two threads with small object caches. No load balancer. Varnish cache is file-based.

sample-medium.yml

> Targets a server with 2GB of memory and two CPU cores. Sets up two ZEO clients, each with one thread with a medium object cache. Uses load balancer to manage the queue to the ZEO clients. Varnish cache is memory-based.

sample-multiserver.yml

> A configuration that demonstrates how to run multiple Zope/Plone installs with different versions and virtual hosting.

The first four samples are meant to be immediately useful. Just copy and customize. The multiserver sample is just a demonstration of several customization techniques. Read it for examples, but don't expect to use it without substantial customization.

Why no `sample-large.yml`? Because a larger server installation is always going to require more thought and customization. We'll discuss those customization points later. The `sample-medium.yml` file will give you a starting point.

Tests

You'll find a `tests.py` program file and a `tests` directory. The `tests` directory contains Doctest files to test our sample configurations. You may add your own.

The `tests.py` program is a convenience script that will run one or more of the Vagrant boxes against one or more of the Doctest files. Run it with no command line argument for usage help. Or, read the source ;)

Basic use of the playbook

Local configuration file

For a quick start, copy one of the `sample-*.yml` files to `local-configure.yml`. The `local-configure.yml` file is automatically included in the main playbook if it's found.

```
cp sample-small.yml local-configure.yml
```

Now, edit the `local-configure.yml` file to set some required variables:

admin_email

The server admin's email. Probably yours. This email address will receive system notices and log analysis messages.

plone_initial_password

The initial administrative password for the Zope/Plone installation. Not the same as the server shell login.

muninnode_query_ips

Are you going to run a Munin monitor on a separate machine? (And, if not, why not?) Specify the IP address of the monitor machine. Or ...

install_muninnode

Remove the "#" on the `install_muninnode:  no` line if you are not using a Munin monitor.

You're also nearly certainly going to want to specify a Plone version via the `plone_version` setting. You should be able to pick any version from 4.3.x or 5.x.x. Note that the value for this variable must be quoted to make sure it's interpreted as a string.

Use with Vagrant

If you've installed Vagrant/Virtualbox, you're ready to test. Since Vagrant manages the connection, you don't need to create a inventory file entry.

There is a Vagrant setup file, `Vagrantfile`, included with the playbook, so you may just open a command-line prompt, make sure your Ansible virtualenv is activated, and type:

```
vagrant up
```

Note: The first time you use a "box" it will be downloaded. These are large downloads; expect it to take some time.

Note: Instructor note: Having several students simultaneously downloading a virtualbox over wifi or a slow connection is a nightmare. Have a plan.

Once you've run **`vagrant up`**, running it again will not automatically provision the virtualbox. In this case, that means that Ansible is not run. So, if you change your Ansible configuration, you'll need to use:

```
vagrant provision
```

Note: When you run `up` or `provision`, watch to make sure it completes successfully. Note that failures for particular plays do not mean that Ansible provisioning failed. The playbook has some tests that fail if particular system features are unavailable. Those test failures are ignored and the provisioning continues. The provisioning has only failed if an error causes it to stop.

An example of an ignored failure:

```
TASK [varnish : Using systemd?] **************************************************
fatal: [trusty]: FAILED! => {"changed": true, "cmd": "which systemctl && systemctl is-
↪enabled varnish.service", "delta": "0:00:00.002085", "end": "2016-09-14 17:50:06.
↪385887", "failed": true, "rc": 1, "start": "2016-09-14 17:50:06.383802", "stderr": "
↪", "stdout": "", "stdout_lines": [], "warnings": []}
...ignoring
```

Vagrant ports

The Vagrant setup (in `Vagrantfile`) maps several ports on the guest machine (the virtualbox) to the host box. The general scheme is to forward a host port that is 1000 greater than the guest port. For example, the load-balancer monitor port on the guest server is `1080`. On the host machine, that's mapped by ssh tunnel to 2080. So, we may see the haproxy monitor at `http://localhost:2080/admin`.

The guest's http port (80) is reached via the host machine's port 1080 – but that isn't actually very useful due to URL rewriting for virtual hosting. If you take a look at `http://localhost:1080` from your host machine, you'll see the default Plone site, but stylesheets, javascript and images will all be missing. Instead, look at the load-balancer port (8080 on the guest, 9080 on the host) to see your ZODB root.

Some quick Vagrant

```
vagrant up                   # bring up the virtualbox
vagrant provision            # provision the virtualbox
vagrant up --no-provision    # bring the box up without provisioning
vagrant halt                 # stop and save the state of the virtualbox
vagrant destroy              # stop and destroy the box
vagrant ssh                  # ssh to the guest box
```

To each of the these commands, you may add an id to pick one of the boxes defined in Vagrantfile. Read Vagrantfile for the ids. For example, `centos7` is the id for a CentOS box.

```
vagrant up centos7
```

Run against cloud

Let's provision a cloud server. Here are the facts we need to know about our cloud server:

hostname

A new server may or may not have a DNS host entry. If it does, use that hostname. If not, invent one and be prepared to supply an IP address.

login id

The user id of a system account that is either the superuser (root) or is allowed to use **sudo** to issue arbitrary commands as the superuser.

password

If your cloud-hosting company does not set up the user account for ssh-keypair authentication, you'll need a password. Even if your account does allow passwordless login, it may still require a password to run **sudo**.

If your cloud-hosting company sets up a root user and password, it's a good practice to login (or use Ansible) to create a new, unprivileged user with sudo rights. Cautious sysadmins will also disable root login via ssh.

connection details

If you don't have a DNS host record for your server, you'll need to have its IP address. If ssh is switched to an alternate port, you'll need that port number.

With that information, create an inventory file (if none exists) and create a host entry in it. We use `inventory.cfg` for an inventory file. A typical inventory file:

```
www.mydomain.co.uk ansible_host=192.168.1.1 ansible_user=steve
```

You may leave off the `ansible_host` if the name supplied matches the DNS host record. You may leave off the `ansible_user` if your user id is the same on the server.

An inventory file may have many entries. You may run Ansible against one, two, all of the hosts in the inventory file, or against alias groups like "plone-servers". See Ansible's inventory documentation for information on grouping host entries and for more specialized host settings.

Now, let's make things easier for us going forward by creating an `ansible.cfg` file in our playbook directory. In that text file, specify the location of your inventory file:

```
[defaults]
inventory = ./inventory.cfg
roles_path = ./roles
```

Smoke test

Now, let's see if we can use Ansible to connect to the remote machine that we've specified in our inventory.

Does the new machine allow an ssh key login, then you ought to be able to use the command:

```
ansible www.mydomain.co.uk -a "whoami"
```

If you need a password for login, try:

```
ansible www.mydomain.co.uk -a "whoami" -k
```

And, if that fails, ask for verbose feedback from Ansible:

```
ansible www.mydomain.co.uk -a "whoami" -k -vvvv
```

Now, let's test our ability to become superuser on the remote machine. If you have passwordless sudo, this should work:

```
ansible www.mydomain.co.uk -a "whoami" -k --become
# omit the "-k" if you need no login password.
```

If sudo requires a password, try:

```
ansible www.mydomain.co.uk -a "whoami" -k --become -K
# again,  omit the "-k" if you need no login password.
```

If all that works, congratulations, you're ready to use Ansible to provision the remote machine.

Note: The "become" flag tells Ansible to carry out the action while becoming another user on the remote machine. If no user is specified, we become the superuser. If no method is specified, it's done via **sudo**.

You won't often use the `--become` flag because the playbooks that need it specify it themselves.

Diagnosing ssh connection failures

If Ansible has trouble connecting to the remote host, you're going to get a message like:

```
myhost | UNREACHABLE! => {
    "changed": false,
    "msg": "Failed to connect to the host via ssh.",
    "unreachable": true
}
```

If this happens to you, try adding `-vvv` to the **ansible** or **ansible-playbook** command line. The extra information may – or may not – be useful.

The real test is to use a direct ssh login in order to get the ssh error. There's a pretty good chance that the identity of the remote host will have changed, and ssh will give you a command line to clean it up.

Running the playbook

We're ready to run the playbook. Make sure you're logged to your ansible-playbook directory and that you've activated the Python virtualenv that includes Ansible.

If you're targetting all the hosts in your inventory, running the playbook may be as easy as:

```
ansible-playbook playbook.yml
```

If you need a password for ssh login, add `-k`.

If you need a password for sudo, add `-K`.

If you need a password for both, add "-k -K".

If you want to target a particular host in your inventory, add `--limit=hostname`. Note that the `--limit` parameter is a search term; all hostnames matching the parameter will run.

Note: As with Vagrant, check the last message to make sure it completes successfully. When first provisioning a server, timeout errors are more likely. If you have a timeout, just run the playbook again. Note that failures for particular plays do not mean that Ansible provisioning failed.

Firewalling

Running the Plone playbook does not set up server firewalling. That's handled via a separate playbook, included with the kit. We've separated the functions because many sysadmins will wish to handle firewalling themselves.

If you wish to use our firewall playbook, just use the command:

```
ansible-playbook firewall.yml
```

`firewall.yml` is just a dispatcher. Actual firewall code is in the `firewalls` subdirectory and is platform-specific. `ufw` is used for the Debian-family; `firewalld`

The general firewall strategy is to block everything but the ports for ssh, http, https and munin-node. The munin-node port is restricted to the monitor IP you specify.

Note: This strategy assumes that you're going to use ssh tunnelling if you need to connect to other ports.

In operation

Hopefully, you've got a provisioned server. Do a quick check by ssh'ing to the server. You should see a welcome message like:

```
This server is configured via Ansible.
Do not change configuration settings directly.

Admin email: steve@dcn.org
Custom Services/Ports
zeoserver: /usr/local/plone-5.0/zeoserver
    /Plone: myhost [u'default']
    zeo server: 127.0.0.1:8100
    haproxy front end: 8080
    zeo clients: 127.0.0.1:8081 127.0.0.1:8082
haproxy monitor: 127.0.0.1:1080
varnish: 127.0.0.1:6081
varnish admin: 127.0.0.1:6082
postfix: 25 (host-only)
nginx:
- myhost: *:80
```

This gives you a list of all the long-lived services installed by the playbook and the interface/ports at which they're attached.

Note the service addresses which begin with `127.0.0.1`. Those services should only answer requests from the server itself: from the localhost. See the firewalling section below for help on tightening this up.

So, how do you connect to local-only ports. Use ssh tunnels.

```
ssh ubuntu@54.244.201.44 -L 1080:localhost:1080 -L 6081:localhost:6081 -L 8080:
↪localhost:8080
```

This is a pretty typical login that creates handy tunnels between ports on your local machine with matching haproxy-admin, varnish and haproxy front-end ports on the remote server.

While you're logged in, check out the status of the **`supervisor`** process-control system, which is used to launch your Zope/Plone processes.

```
sudo supervisorctl status
```

will list all the processes controlled by supervisor.

Plone setup and directories

While you're logged in, let's take a look at the Plone/Zope setup.

You may modify the Zope/Plone directory layout created by the playbook. Unless you do, the Playbook will put Plone's programs and configuration files in `/usr/local` by Plone version. Data files will be in `/var/local`. This split is intended to make it easier to organize backups and to put data on a different physical or logical device.

Unless you change it, backups are also under `/var/local`. It's easy to change this, and it's not a bad idea to have backups on a different device.

In terms of file ownership and permissions, the Playbook pretty much follows the practices of the Plone Unified Installer. Program and configuration files are owned by the `plone_buildout` user, and data, log and backup files are owned by the `plone_daemon` user. A `plone_group` is used to give some needed communication, particularly the ability of buildout to create directories in the data space.

This means that if you need to run `bin/buildout` via login, it must be run as the `plone_buildout` user.

```
sudo -u plone_buildout bin/buildout
```

Typically, you would never start the main ZEO server or its clients directly. That's handled via **`supervisorctl`**. There's one exception to this rule: the playbook creates a ZEO client named `client_reserved` that is not part of the load-balancer pool and is not managed by supervisor. The purpose of this extra client is to allow you to handle run scripts or debug starts without affecting the load-balanced client pool. It's particularly a good idea to use this mechanism to test an updated buildout:

```
sudo -u plone_daemon bin/client_reserved fg
```

Restart script

Still logged in? Let's take a look at another part of the install: the restart script. Look in your buildout directory for the scripts directory. In it, you should find `restart_clients.sh`. (Go ahead and log out if you're still connected.)

This script, which needs to be run as the superuser via **`sudo`**, is intended to manage hot restarts. Its general strategy is to run through your ZEO clients, sequentially doing the following:

1. Mark it down for maintenance in haproxy;
2. stop client;
3. start client; wait long enough for it to start listening
4. Fetch the homepage directly from the client to load the cache. This will be the first request the client receives, since haproxy hasn't have marked it live yet. So, when haproxy marks it live, the cache will be warm.
5. Mark the client available in haproxy.

After running through the clients, it flushes the varnish cache.

This is only really useful if you're running multiple ZEO and using haproxy for your load balancer.

Client logs

Unless you change it, the playbook sets up the clients to maintain 5 generations of event and access logs. Event logs are rotated at 5MB, access logs at 20MB.

cron jobs

The playbook automatically creates **cron** jobs for ZODB backup and packing. These jobs are run as plone_daemon.

The jobs are run in the early morning in the server's time zone. Backup is run daily; packing weekly.

Load balancing

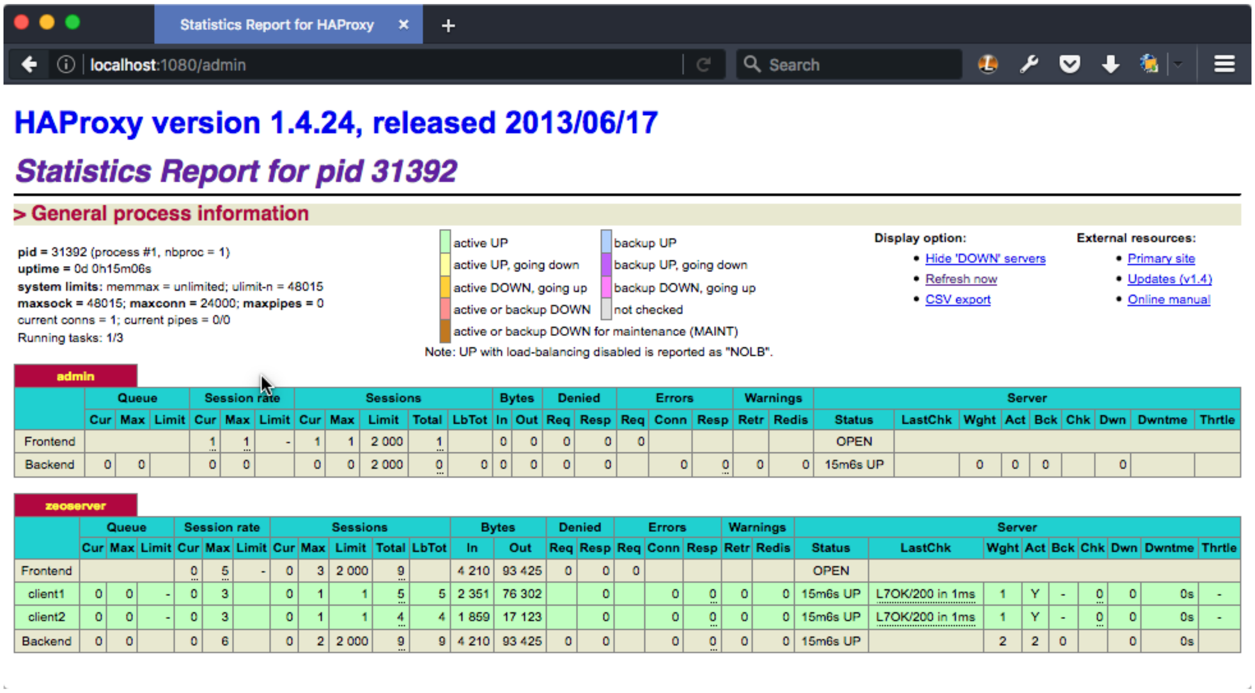
Let's step up the delivery stack.

All but the smallest sample playbooks set up ZEO load balancing via haproxy. One of the things we gain from haproxy is good reporting.

The web interface for the haproxy monitor is deliberately not available to a remote connection. It's easy to get around that with an ssh tunnel:

```
ssh ubuntu@ourserver -L 1080:localhost:1080
```

Now we may ask for the web report at http://localhost:1080/admin. Since we're restricting access, we don't bother with a password.



HAProxy version 1.4.24, released 2013/06/17

Statistics Report for pid 31392

> General process information

pid = 31392 (process #1, nbproc = 1)
uptime = 0d 0h15m06s
system limits: memmax = unlimited; ulimit-n = 48015
maxsock = 48015; **maxconn** = 24000; **maxpipes** = 0
current conns = 1; current pipes = 0/0
Running tasks: 1/3

🟩	active UP	▨	backup UP
🟨	active UP, going down	▨	backup UP, going down
🟧	active DOWN, going up	▨	backup DOWN, going up
🟥	active or backup DOWN	▨	not checked
🟫	active or backup DOWN for maintenance (MAINT)		

Note: UP with load-balancing disabled is reported as "NOLB".

Display option:
- Hide 'DOWN' servers
- Refresh now
- CSV export

External resources:
- Primary site
- Updates (v1.4)
- Online manual

admin

	Queue			Session rate			Sessions					Bytes		Denied		Errors			Warnings		Server								
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle
Frontend				1	1	-	1	1	2 000	1		0	0	0	0	0					OPEN								
Backend	0	0		0	0		0	0	2 000	0	0	0	0	0	0		0	0	0	0	15m6s UP		0	0	0		0		

zeoserver

	Queue			Session rate			Sessions					Bytes		Denied		Errors			Warnings		Server								
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle
Frontend				0	5	-	0	3	2 000	9		4 210	93 425	0	0	0					OPEN								
client1	0	0	-	0	3		0	1	1	5	5	2 351	76 302		0		0	0	0	0	15m6s UP	L7OK/200 in 1ms	1	Y	-	0	0	0s	-
client2	0	0	-	0	3		0	1	1	4	4	1 859	17 123		0		0	0	0	0	15m6s UP	L7OK/200 in 1ms	1	Y	-	0	0	0s	-
Backend	0	0		0	6		0	2	2 000	9	9	4 210	93 425	0	0		0	0	0	0	15m6s UP		2	2	0		0	0s	

Fig. 2.3: Haproxy monitor at http://localhost:1080/admin

If your optimizing, it's a great idea to look at the haproxy stats to see what kind of queues are building up in your ZEO client cluster.

A word about the cluster strategy. We set up our clients with a single ZODB connection thread. There's a trade-off here. Python's threading isn't great on multi-core machines. If you've got only one CPU core available, that's fine. But modern servers typically have several cores; this scheme allows us to keep those cores more busy than they would be otherwise. The cost is somewhat more memory use: a ZEO client with multiple threads does some memory sharing between threads. It's not a lot, but that gives it some memory use advantage over multiple, single-threaded clients. You may want to make that trade off differently.

We also have haproxy set up to only make one connection at a time to each of our ZEO clients. This is also a trade off. We lose the nice client behavior of automatically using different delivery threads for blobs. But, we lower the risk that a request will sit for a long time in an individual client's queue (the client's connection queue, note haproxy's). If someone makes a request that will take several seconds to render and return, we'd like to avoid slowing down the response to other requests.

Reverse-proxy caching

We use Varnish for reverse-proxy caching. The size of the cache and its storage strategy is customizable.

By default, we set up 512MB caches. That's probably about right if you're using a CDN, but may be low if if your site is large and you're not using a CDN. The two small samples use Varnish's `file` method for cache storage. The larger samples use `malloc`.

Varnish's control channel is limited to use by localhost and has no secret.

In a multi-Plone configuration, where you set up multiple, separate Plone servers with separate load-balancing front ends, our VCL setup does the dispatching to the different front ends.

Web hosting

We use nginx for the outer web server, depending on it to do efficient URL rewriting for virtual hosting and for handling https.

We'll have much more to say about virtual hosting later when we talk about how to customize it. What you need to know now is that simple virtual hosting is automatically set up between the hostname you supply in the inventory and the `/Plone` site in the ZODB. So, you should be able to immediately ask for your server via http and get a Plone welcome page.

If your inventory hostname does not have a matching DNS host record, you're going to see something like:

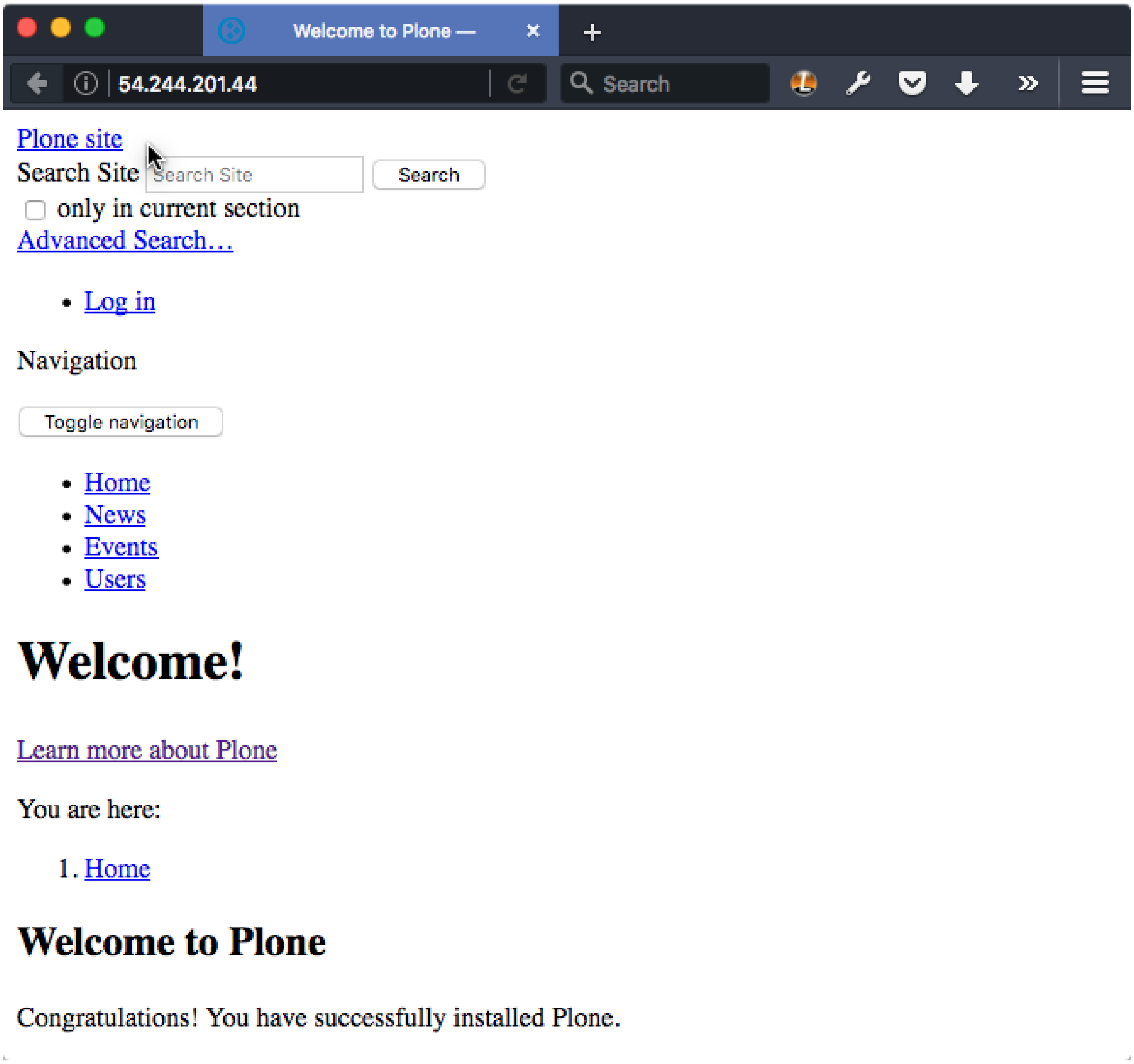
You're seeing a virtual-hosting setup error. The requested *page* is being returned, but all the resource URLs in the page – images, stylesheets and javascript resources – are pointing to the hostname supplied in the inventory. You may fix that by supplying a DNS-valid hostname, or by setting up specific virtual hosting. That's detailed below.

That's it for the delivery stack. Let's explore the other components installed by the playbook.

Postfix

We use Postfix for our mailhost, and we set it up in a send-only configuration. In this configuration, it should not accept connections from the outside world.

Note: You will probably have another SMTP agent that's the real mail exchange (MX) for your domain. Make sure that server is configured to accept mail from the `FROM` addresses in use on your Plone server. Otherwise, mail exchanges that "grey list" may not accept mail from your Plone server.



Plone site

Search Site Search Site Search

□ only in current section

Advanced Search…

- Log in

Navigation

Toggle navigation

- Home
- News
- Events
- Users

Welcome!

Learn more about Plone

You are here:

1. Home

Welcome to Plone

Congratulations! You have successfully installed Plone.

Fig. 2.4: Typical virtual hosting error.

Updating system packages

On Debian family Linux, the playbook sets up the server for automatic installation of routine updates. We do not, however, set up an automatic reboot for updates that require a system restart. So, be aware that you'll need to watch for "reboot required" messages and schedule a reboot.

fail2ban

On Debian family Linux, the playbook installs `fail2ban` and configures it to temporarily block IP addresses that repeatedly fail login attempts via ssh.

Monitoring

`logwatch` is installed and configured to email daily log summaries to the administrative email address.

Unless you prevent it, **`munin-node`** is installed and configured to accept connections from the IP address you designate. To make use of it, you'll need to install **`munin`** on a monitoring machine.

The **`munin-node`** install by the playbook disables many monitors that are unlikely to be useful to a mostly dedicated Plone servers. It also installs a Plone-specific monitor that reports resident memory usage by Plone components.

Changes philosophy

The general philosophy for playbook use is that you make all server configuration changes via Ansible. If you find yourself logging in to change settings, think again. That's the road to having a server that is no longer reproducible.

If you've got a significant change to make, try it first on a test server or a Vagrant box.

This does not mean that you'll never want to log into the server. It just means that you shouldn't do it to change configuration.

More customized use

We intend that you should be able to make most changes by changing default variable settings in your `local_configure.yml` file. We've made a serious effort to make sure that all those settings are documented in the *Plone's Ansible Playbook <http://docs.plone.org/external/ansible-playbook/docs/index.html>* documentation.

For example, if you want to change the time at which backup occurs, you can check the doc and discover that we have a plone-backup-at setting. The default setting is:

```
plone_backup_at:
  minute: 30
  hour: 2
  weekday: "*"
```

That's 02:30 every morning.

To make it 03:57 instead, use:

```
plone_backup_at:
  minute: 57
  hour: 3
  weekday: "*"
```

in your `local_configure.yml` file.

Common customization points

Let’s review the settings that are very commonly changed.

Plone setup

Eggs and versions

You’re likely to want to add Python packages to your Plone installation to enable add-on functionality.

Let’s say you want to add Products.PloneFormGen and webcouturier.dropdownmenu. Just add to your local_configure.yml:

```
plone_additional_eggs:
    - Products.PloneFormGen
    - webcouturier.dropdownmenu
```

If you add eggs, you should nearly always specify their versions:

```
plone_additional_versions:
  - "Products.PloneFormGen = 1.7.16"
  - "Products.PythonField = 1.1.3"
  - "Products.TALESField = 1.1.3"
```

That takes care of packages that are available on the Python Package Index. What if your developing packages via git?

```
plone_sources:
  -  "some.other.package = git git://example.com/git/some.other.package.git rev=1.1.5"
```

There’s more that you can do with the plone_sources setting. See the docs!

buildout from git repo

It’s entirely possible that the buildout created by the playbook won’t be adequate to your needs. If that’s the case, you may check out your whole buildout directory via git:

```
buildout_git_repo: https://github.com/plone/plone.com.ansible.git
buildout_git_version: master
```

Make sure you check the documentation on this setting. Even if you use your own buildout, you’ll need to make sure that some of the playbook settings reflect your configuration.

Running buildout and restarting clients

By default, the playbook tries to figure out if **buildout** needs to be run. If you add an egg, for example, the playbook will run buildout to make the buildout-controlled portions of the installation update.

If you don’t want that behavior, change it:

```
plone_autorun_buildout: no
```

If autorun is turned off, you’ll need to log in to run buildout after it completes the first time. (When you first run the playbook on a new server, buildout will always run.)

If automatically running buildout bothers you, automatically restarting Plone after running buildout will seem foolish. You may turn it off:

```
plone_restart_after_buildout: no
```

That gives you the option to log in and run the client restart script. If you're conservative, you'll first try starting and stopping the reserved client.

Note: By the way, if buildout fails, your playbook run will halt. So, you don't need to worry that an automated restart might occur after a failed buildout.

Web hosting options

It's very likely that you're going to need to make some changes in nginx configuration. Most of those changes are made via the `webserver_virtualhosts` setting.

`webserver_virtualhosts` should contain a list of the hostnames you wish to support. For each one of those hostnames, you may make a variety of setup changes.

The playbook automatically creates a separate host file for each host you configure.

Here's the default setting:

```
webserver_virtualhosts:
  - hostname: "{{ inventory_hostname }}"
    default_server: yes
    zodb_path: /Plone
```

This connects your inventory hostname for the server to the /Plone directory in the ZODB.

A more realistic setting might look something like:

```
webserver_virtualhosts:
  - hostname: plone.org
    default_server: yes
    aliases:
      - www.plone.org
    zodb_path: /Plone
    port: 80
    protocol: http
    client_max_body_size: 4M
  - hostname: plone.org
    zodb_path: /Plone
    address: 92.168.1.150
    port: 443
    protocol: https
    certificate_file: /thiscomputer/path/mycert.crt
    key_file: /thiscomputer/path/mycert.key
```

Here we're setting up two separate hosts, one for http and one for https. Both point to the same ZODB path, though they don't have to. The https host item also refers to a key/certificate file pair on the Ansible host machine. They'll be copied to the remote server.

Alternatively, you could specify use of certificates already on the server:

```
webserver_virtualhosts:
  - hostname: ...
    ...
    certificate:
      key: /etc/ssl/private/ssl-cert-snakeoil.key
      crt: /etc/ssl/certs/ssl-cert-snakeoil.pem
```

Caution: One hazard for the current playbook web server support is that it does **not** delete old host files. So, if you had previously set up `www.mynewclient.com` and then deleted that item from the playbook host list, the nginx host file would remain. Log in and delete it if needed. Yes, this is an exception to the "don't login to change configuration rule".

Extra tricks

There are a couple of extra setting that allow you to do extra customization if you know nginx directives. For example:

```
- hostname: plone.com
  protocol: http
  extra: return 301 https://$server_name$request_uri;
```

This is a *redirect to https*. It takes advantage of the fact that if you do not specify a zodb_path, the playbook will not automatically create a location stanza with a rewrite and proxy_pass directives.

Mail relay

Some cloud server companies do not allow servers to directly send mail to standard mail ports. Instead, they require that you use a *mail relay*. This is a typical setup:

```
mailserver_relayhost: smtp.sendgrid.net
mailserver_relayport: 587
mailserver_relayuser: yoursendgriduser
mailserver_relaypassword: yoursendgridpassword
```

Bypassing components

Remember our stack diagram? The only part of the stack that you're stuck with is Plone. All the other components my be replaced. To replace them, first prevent the playbook from installing the default component. Then, use a playbook of your own to install the alternative component.

For example, to install an alternative to the Postfix mail agent, just add:

```
install_mailserver: no
```

Note: If you choose not to install the haproxy, varnish or nginx, you take on some extra responsibilities. You're going to need to make sure in particular that your port addresses match up. If, for example, you replace haproxy, you will need to point varnish to the new load-balancer's frontend. You'll need to point the new load balancer to the ZEO clients.

Multiple Plones per host

So far, we've covered the simple case of having one Plone server installed on your server. In fact, you may install additional Plones.

To do so, you create a list variable `playbook_plones` containing all the settings that are specific to one or more of your Plone instances.

Nearly all the plone_* variables, and a few others like loadbalancer_port and webserver_virtualhosts may be set in playbook_plones. Here's a simple example:

```
playbook_plones:
  - plone_instance_name: primary
    plone_zeo_port: 8100
    plone_client_base_port: 8081
    loadbalancer_port: 8080
    webserver_virtualhosts:
      - hostname: "{{ inventory_hostname }}"
        aliases:
          - default
        zodb_path: /Plone
  - plone_instance_name: secondary
    plone_zeo_port: 7100
    plone_client_base_port: 7081
    loadbalancer_port: 7080
    webserver_virtualhosts:
      - hostname: www.plone.org
        zodb_path: /Plone
```

Note that you're going to have to specify a minimum of an instance name, a zeo port and a client base port (the address of client1 for this Plone instance.)

You may specify up to four items in your `playbook_plones` list. If you need more, see the docs as you'll need to make a minor change in the main playbook.

The Plone Role – using it independently

Finally, for really big changes, you may find that the full playbook is of little or no use. In that case, you may still wish to use Plone's Ansible Role independently, in your own playbooks. The Plone server role is maintained separately, and may become a role in your playbooks if it works for you.

Maintenance strategies

This section covers strategies for long-run maintenance of your playbook. If you're successful with Plone's Ansible Playbook, you will wish to keep an eye on its continued development. You may wish to be able to integrate bug fixes and new features that have become part of the distribution. But, since this project targets production servers, you'll wish to be very careful in integrating those changes so that you minimize risk of breaking a live server configuration.

Caution: Rule 1: If it changes, test it.

Using Ansible (or other configuration-management systems) makes it easier to test a whole server configuration. Make use of that fact! You may test by running your playbook against a Vagrant box or against a staging server.

Make sure your test server matches the current live configuration. Copy backup Plone data from the live server; restore it on the test server. Then, make your changes in the playbook (or its Ansible support) and run it against the test server. Only on testing success should you run against the live server.

Virtualenv

If you followed our installation instructions, you have a Python virtualenv attached to your playbook checkout. That virtualenv has its own installation of Ansible. That's good, because it protects your playbook against unexpected changes in the global environment – such as Ansible being updated by the OS update mechanisms.

You may need or wish to update the installation of Ansible in your Virtualenv. If so, make sure you use the copy of **pip** in your virtualenv. Then, test running your playbook with your new Ansible.

What belongs to the playbook and what doesn't

The general strategy for playbook changes is to not modify anything that's included with the playbook. We've gone to some trouble to make sure that you can make most forseeable setup changes without touching distribution files.

The `local-configure.yml` is an example of this strategy. It is **not** included with the distribution files. It never will be. We will also never include an `inventory.cfg` file.

That means that you may safely merge changes from the STABLE branch of https://github.com/plone/ansible-playbook without fear of overwriting those files. You may also create new playbooks; just give them different names. The extra playbooks might handle installs of extra components, firewalling, user setup, whatever.

Git forks

But, what if you want to use version control with your own added files?

In this case, you will wish to *fork* https://github.com/plone/ansible-playbook. Add your extra files to those included with your local checkout of the git fork and push upstream to your git repository. Then, occasionally merge changes from the Plone github account's repository into your fork, typically by rebasing from Plone's upstream repository STABLE branch. Make sure you keep your added files when you do so.

Maintenance strategies – multiple hosts

The `local-configure.yml` file strategy makes it easy to get going with Plone's playbook fast. But it breaks down if you wish to maintain multiple, different hosts with the playbook. Fortunately, there's an easy way to handle the problem.

Create a `host_vars` directory inside your playbook directory (the one containing playbook.yml). Now, inside that directory, create one file per target host, each with a name that matches the inventory entry for the host, plus `.yml`. Each of these files should be the same as the local-configure.yml file that would be used if this was a single host. Delete the no-longer-needed `local-configure.yml` file.

See also:

http://docs.plone.org/manage/deploying/

OpsWorks

Orchestrating Plone Deployments with Amazon OpsWorks Using Amazon Opsworks to orchestrate clusters for scalable/high availablity deployments.

Orchestrating Plone Deployments with Amazon OpsWorks

Note: This training is meant to be used in a course or read and worked through by an individual user. Instructors should note that this makes it more discursive than it would be if it was only meant for classroom use. Many sections may be zipped through in a class, noting to students that the full text is available for review.

Contents:

Introduction

The subject of this training is using Amazon Opsworks deployment system to orchestrate complex, scalable, and redundant multi-server deployments of Plone. The tools presented herein provide a mechanism for generically defining server requirements and resources to launch fully configured Amazon EC2 instances running Plone in a coordinated distributed manner.

Amazon Opsworks does not provide the flexibility of Ansible deployments. It is tied to Amazon cloud infrastructure, and is only fully tested for servers running Ubuntu LTS. It does provide an unique infrastructure to automate communication among multiple servers, allowing automated discovery and inclusion of resources, and facilitating features like auto-scaling and auto- healing.

Opsworks is built on Chef, which is a configuration management system similar to Ansible, but built on Ruby *[0]. The tools and concepts described here attempt to ensure that you can deploy a complex Plone site without having to learn any Chef or Ruby.

Deployment Terminology

It's probably a good idea to be familar with a few core Chef concepts, though digging deeply into Chef is definitely not something I encourage Python developers to do.

- `Resource`: The basic building block in Chef (and also Ansible); defines files, directories, installed packaes, services, etc.
- `Recipe`: A collection of resource definitions with some logic to connect them. These can be very simple or extraordinarily complex; A recipe can depend on other recipes. These basically play the same role as `Tasks` in Ansible.
- `Cookbook`: A collection of recipes required to setup a service or similar. These play a similar role to `Roles` in Ansible. These generally can be found in the Chef Supermarket like Roles from the Ansible Galaxy.
- `Berkshelf`: A single file configuration defining the set of cookbooks needed for a deployment. It consists of a `Berksfile` which defines locations and versions of all cookbooks required for a deployment.
- `Attributes`: The deployment specific configuration for the cookbook and recipes. This is essentially a collection of JSON like primitives, similar to YAML group/host `Vars`.

Opsworks

Amazon Opsworks takes this basic configuration framework and provides its own set of concepts, to implement cluster orchestration. When using Opsworks, you will be making use built-in Opsworks Chef Cookbooks provided by

[0] Yuck!

Amazon. These built-in Cookbooks provide a number of Recipes for configuring and deploying many types of applications using simple TTW configuration from the Opsworks control panel. These include Node.js, Rails, PHP, and Java applications, but not Python *[0].

I've created a couple supplemental Cookbooks that extend the existing Opsworks deployment recipes to support Python and Plone along with other supporting services that are useful when making production deployments of Plone.

Opsworks has its own vocabulary of concepts related to deploying and orchestrating clusters of servers. The building blocks of Opsworks are:

- `Stack`: The fundamental container for your configuration, this lives in a particular EC2 region and contains all the configuration for your cluster. Typically you would have a separate production stack and development stack. Creating this is the first step in the process of defining your cluster. Stacks can be cloned to replicate configuration across regions.
- `Layers`: A Layer defines a discrete set of functionality that may be provided by a server Instance. For example, a Plone cluster may have a front end Layer running an Nginx web server, Varnish proxy cache and HAProxy load balancer †[0] , an Application Layer for your ZEO client instances, an Application Layer for your ZEO server, and a maintenance layer to manage database backups and packs. Layers define what recipes will be run on an instance, and which OS packages it requires, along with any Amazon resources and permssions are required to provide a service (e.g. static IP addresses, additional EBS storage volumes).
- `Instances`: An OpsWorks Instance is similar to an EC2 instance, it has a type (e.g. from micro to xlarge), an OS and an Availability Zone, but it is an abstraction. It becomes an actual EC2 instance once it's been started, but before that it's simply a metadata about a desired server. Instances are assigned to one or more Layers, and come in three varieties, 24/7, time-based and load-based.
- `Apps`: An App points to a code repository (in our case a buildout) which you want to deploy to a specific Application Layer. Typically you would have an App for your Plone instances and another for your Zeoserver. Both these Apps would typically point to the same buildout repository. You might also create an App to configure a Plone specific Solr server or to run a additional applications within the cluster.
- `Resources`: A set of Amazon EC2 resources that will be used by the stack by being attached/assigned to Instances when they are started. These include Elastic IP addresses, EBS storage volumes and RDS databases (useful you are running Relstorage).

A Stack can be configured with a single Instance running all the Layers, or multiple Instances each running different Layers. You might, for example, have a production stack with five Instances running the Plone ZEO client Application Layer, a single instance running the ZEO server Application Layer, and two Instances running the front end proxy/loadbalancer Layer (with an Elastic Load Balancer in front of those). You might also have a staging stack with all the same Layers applied to a single modest server. Other than the Instance definitions (and perhaps the App repository branch), these Stacks would be essentially identical.

Instance Lifecycle

Each Opsworks Instance goes through a few phases during its lifecycle:

- `setup`
- `deploy`
- `configure`
- `undeploy`
- `shutdown`

[0] Boo!

[0] Though you could separate each of these front end services into their own layers if you really wanted to, we combine them by default under a customized HAProxy layer which already provides a nice UI for a few HAProxy features.

Each of these lifecycle phases runs recipes assigned to that phase in the assigned Layers. When these recipes are run, the Stack configuration is passed to the server. This configuration includes complete information about the state of the entire Stack and all of its running Instances.

When an Instance starts, it first goes through a `setup` phase: installing all package dependencies for all assigned Layers and running all the recipes assigned to the `setup` phase of those Layers.

Once the `setup` phase is complete, a `deploy` phase is automatically started. running all the recipes assigned to the `deploy` phase of any assigned Layers. Subsequently, you may manually run a `deploy` for a specific Application on any or all of the instances to update the application code and reconfigure services.

The `shutdown` phase is run automatically before an instance is stopped.

The `undeploy` phase is rarely used. It is triggered when an application is manually removed from an instance.

Whenever an Instance is started or stopped and it's `setup` or `shutdown` phase has completed a `configure` phase is initiated on all running instances. As with all recipe runs, the `configure` phase recipes are passed data about all the curently running Instances and their Layers so that they can automatcially reconfigure themselves based on the updated state of the Stack. For example, a load balancer may need to automatically add or remove Plone ZEO clients from it's list of active backends, a ZEO client may need to change its ZEO server or its Relstorage Memcache if configuration for those services have changed.

This `configure` phase, during which the current cluster state is automatically shared with all the instances, is where the orchestration magic happens.

Creating Your First Stack

Setting up a Stack with all of its layers is a tedious excersise it TTW configuration. Thankfully there's another AWS tool (there's always another AWS tool) called CloudFormation that lets us quickly configure a basic Plone stack with the most common layers configured.

If you navigate to CloudFormation in the AWS console you'll be presented with the option to create a `Stack`. Confusingly, a CloudFormation Stack is not the same thing as an Opsworks Stack, but the former is what we use to automate the creation of the latter so we can use the terms a bit interchangeably.

You'll want to download the following file from Github: https://raw.githubusercontent.com/alecpm/opsworks-web-python/master/plone_buildout/examples/zeoserver-stack.template

And use the "Upload a template to Amazon S3" option to upload the above file which provides a basic ZEO server stack configuration *[0]. You may want to select the EC2 region for you stack before creating the stack, but if you don't you can always clone the stack into another region later. The stack creation will take a few minutes; once it succeeds you can navigate to the Opsworks control panel to see your new Stack. †[0]

The CloudFormation setup creates a stack outside of a VPC (Virtual Private Cloud), which is probably not ideal since some instance options are not available outside of a VPC. If you want the stack to use a VPC or to be in a different EC2 region than you initially ran CloudFormation from, then you can clone the Stack from the Opsworks Dashboard to set your desired region and VPC settings.

There are a few important settings which CloudFormation is not able to manage and have to be modified after stack creation. The two Apps (`Plone Instances` and `Zeoserver`) should be edited to set the `Data Source` to `None` (this setting is useful for a Relstorage configuration, but does nothing for a ZEO server configuraiton). Eventually, you will probably want to use your own buildout repository in these App configurations, but any buildout you use should probably be cloned from the one used in this demo configuration because it provides a number of buildout parts and variables that the deployment recipes expect to be in place: https://github.com/alecpm/opsworks_example_buildouts.git

[0] There is also a RelStorage version of this template, though turning a Zeoserver Stack into a Relstorage Stack simply involves deleting the ZEO server Layer and adding a built-in Memcached Layer.

[0] Before creating a CloudFormation Stack you'll be asked to confirm that AWS resources may be created. The stack template here only creates cost-free configuration resources.

Finally, you'll need to provide some additional configuration (Chef Attributes) in the form of the Stack `Custom JSON` which can be edited in the Stack Settings control panel. The following should be a reasonable starting point:

```
{
    "plone_instances" : {
        "nfs_blobs" : true
    },
    "deploy" : {
        "plone_instances" : {
            "buildout_extends": ["cfg/sources.cfg"],
            "buildout_additional_config": "[client1]\nuser=admin:**change-me**"
        },
        "zeoserver" : {
            "buildout_extends": ["cfg/sources.cfg"]
        }
    }
}
```

Note the `buildout_additional_config` attribute, which allows you to insert arbitrary configuration and overrides into the generated buildout `deploy.cfg`. In this case, it's used to set a custom admin password for your new plone instance. There are similar `buildout_parts_to_include` and `buildout_extends` attributes which allow you to customize the parts used for a particular deploy and any additional configuration files to include. For example, typically I will use a include a `cfg/production_sources.cfg` in my production stack which sets revision/tag pins for any external source dependencies in for production deployments.

You may also wish to set the `Opsworks Agent Version` to `Use latest version`, and choose a `Hostname Theme` for fun.

Note that this default configuration uses a blob directory shared over NFS. That's not necessary if you're going to use a single intsance configuration that you plan never to grow (perhaps for a staging server), but if you think you might want multiple servers running ZEO clients, then starting out with a network shared storage for blobs is probably the best way to go. You could also configure shared blobs using the GlusterFS distributed filesystem (this can be tricky and is only recommended if you are already familiar with GlusterFS), S3-fuse Fs (slow), or serve them from the ZEO Server or Relstorage DB. If you do want a single server configuration with no network blob share, then you'll need to add a line of configuration for the blob storage location, e.g.:

```
"plone_blobs": {"blob_dir": "/mnt/shared/blobs/blobstorage"},
```

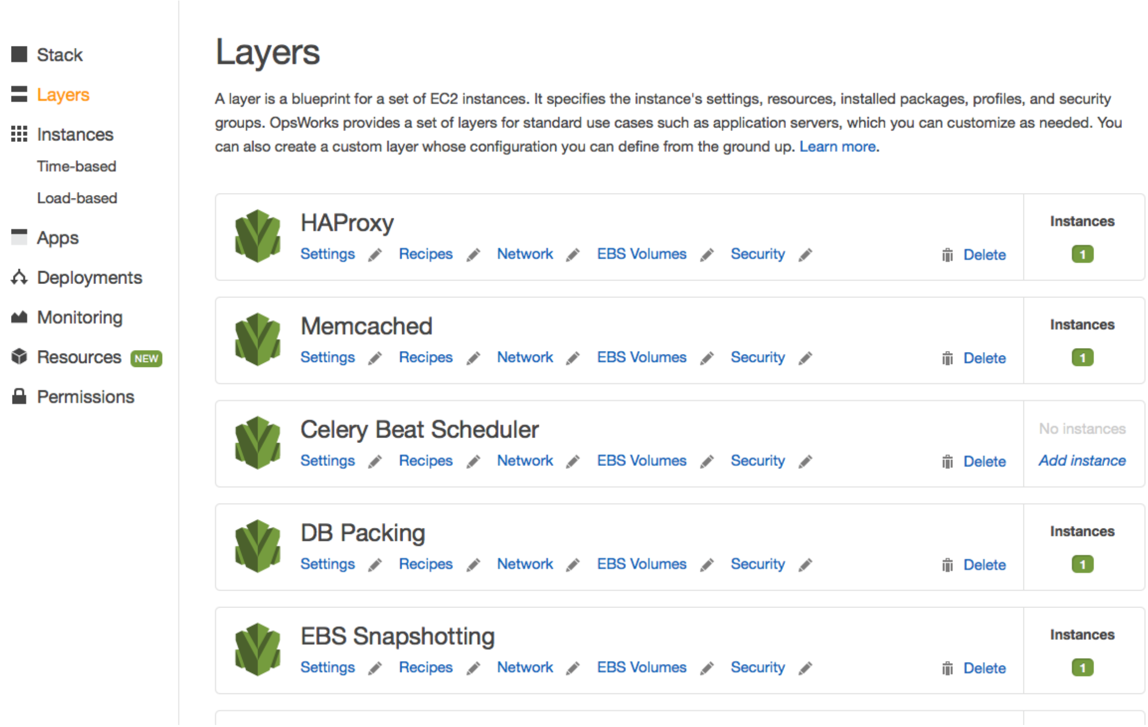
You may also want to change the load balancer stats access password in the HAProxy layer.

By default, each server is protected by a firewall that only allows access to specific services defined by that instances layers. Our layers are heavily customized, so the defaults aren't always sufficient. You'll want to ensure that the servers can all communicate with one another over all desired ports, and you'll probably want to be able to bypass the default firewall from specific externalIP addresses to get direct access to your ZEO Clients, etc. The simplest way to do that is to go to the Security tab for each of the Layers and add the `default` security group to each of them. ‡[0]

Adding an Instance

At this point you can navigate to the `Instances` control panel and create an instance in a particular layer. Once you've defined your first instance you can assign it to additional layers. Once you pick an appropriate instance size (t2.micro is fine for playing around), you should be able to use the instance defaults, though the initial EBS volume size is something you may want to configure later if you don't intend to use separate mount points for data storage.

[0] This could be done with more granularity, but `default` is usually a safe bet. By default, `default` allows servers within your VPC full access to one another, but doesn't permit any outside access. You can configure the `default` security group to allow your personal IPs direct access to any specific ports you may want want to access remotely.



Stack
Layers
Instances
Time-based
Load-based
Apps
Deployments
Monitoring
Resources NEW
Permissions

Layers

A layer is a blueprint for a set of EC2 instances. It specifies the instance's settings, resources, installed packages, profiles, and security groups. OpsWorks provides a set of layers for standard use cases such as application servers, which you can customize as needed. You can also create a custom layer whose configuration you can define from the ground up. Learn more.

HAProxy
Settings Recipes Network EBS Volumes Security Delete
Instances 1

Memcached
Settings Recipes Network EBS Volumes Security Delete
Instances 1

Celery Beat Scheduler
Settings Recipes Network EBS Volumes Security Delete
No instances Add instance

DB Packing
Settings Recipes Network EBS Volumes Security Delete
Instances 1

EBS Snapshotting
Settings Recipes Network EBS Volumes Security Delete
Instances 1

Once you've created the first instance you'll want to add it other layers using the `Existing Opsworks` tabs. You will probably want to skip the `EBS Snapshotting` layer for now, and if you disabled NFS you should skip the `Shared Blobs` layer too.

By default the `Zeoserver` layer and the `Shared Blobs` layer both create and attach EBS volumes to any instances assigned to them (for the filestorage and NFS shared blobstorage respectively). This is optional when using an EBS backed instance with an adequately sized root volume, but is mandatory when using instance store backed instances. Traditionally, instance store backed instances had some performance and cost advantages, but those advantages have largely vanished recently, and EBS instances can stop and start much faster after initial instance creation. For testing you may want to delete the EBS volume resources from those layers before starting your instance.

Note: I still like using instance store instances with sepearate attached EBS volumes because those instances make no promises about retaining configuration changes outside of the explicitly mounted EBS volumes, and that keeps me from twiddling server configuration in ways that might not be repeatable. They also help avoid some I/O concurrency issues you may run into with an all EBS configuration, and allow more straighforward vertical scaling.

Now you should be able to start your instance, and after a little while (depending on the instance size), you will have a server up and running.

This Zope instance won't have a Plone site yet, so having added the `default` security group earlier in order to allow yourself direct access to the ZEO clients will come in handy here. Your instance should have a public IP address (the front end layer assigns an Elastic IP by default, though you could manually transfer one in if you were moving an existing EC2 server). You should be able to access the first ZEO client at port 8081 and create your Plone site.

Caveats

There are a few restrictions on what can and can't be done when in of Opsworks which can occasionally cause annoyance:

- Instances can only be added to layers when the Instances are stopped. So you cannot add additional Layers of functionality to an already running Instance. There are workarounds for this limitation (such as adding recipes or package dependencies to existing layers and re-running the relevant phases), but it can be frustrating.
- You cannot change the security groups of a running instance, and changes to a Layer's security groups don't apply to running instances. Thankfully, any changes to the firewall rules for a security group will affect all running instances in that group. It's best to make sure your Layers assign all the security groups you might need before starting an instance.
- A setup or deploy may fail because of problems accessing Repos or PyPI packages. If the initial instance setup fails, it is not generally necessary to stop, wait and then start the instance (which can take a long time), you generally can re-run the `setup` phase from the Stack panel using the `Run Command` button.
- Downloading public packages from PyPI and dist.plone.org is often the slowest part of initial instance setup. It can help tremendously to have a tarball of all required eggs stored in a public S3 url, you can use the Custom JSON to tell OpsWorks to fetch this tarball before running the buildout. The configuration goes under the `deploy[app_name]` key and looks like [159]

```
"buildout_cache_archives" : [{"url" : "https://my-bucket.s3.amazonaws.com/my-eggs-
↪archive.tgz", "path" : "shared"}]
```

Deploying Changes

Now that you've got one or more Instances up and running, you may need to update the code on them. Traditionally, you'd SSH into the server pull in new changes from the repo(s), run buildout, and restart ZEO clients if necessary. With Opsworks, you click a deploy button and everything is handled automtically.

So what happens when you click the Deploy button for an Opsworks App?

1. The instance looks to see if there's a new revision on the App's buildout repository (accounting for the branch or revision setting in the App configuration).
2. If there are changes to the repository, then it makes a new clone of the repository and puts it in a directory under `releases` named by the checkout timestamp. It then generates a new `deploy.cfg` based on the Stack Configuration, including information about currently running instances and layers. Then it runs bootstrap and buildout with that configuration. If the process succeeds, it symlinks the `release/$timestamp` directory to `current` and restarts the ZEO clients.
3. If there are no changes to the repository, then it generates a new `deploy.cfg` based on the current Stack configuration. If that file differs from the existing deploy.cfg (e.g. because of changes in the Stack's Custom JSON), then it will re-run buildout, and - if the buildout succeeds - restart the ZEO clients.
4. If there are no changes to the repository, and the new `deploy.cfg` is identical to the prior version, then it checks for an `always_buildout_on_deploy` flag in the Stack's `deploy[appname]` Custom JSON. If that flag is true, then it runs buildout and restarts the ZEO clients on success. This is useful if you are deploying changes from external repositories pulled in by mr.developer, even when the buildout repo itself hasn't changed.

Steps 1 and 2 are essentially a `Capistrano` style deployment familar from the Rails world. This process allows for explicit rollback of deployed code to prior versions at any time. Steps 3 and 4 are buildout specific and don't support rollbacks in the same way.

You can run a deploy on a single Instance or on many at once. The deploy will run in parallel on all Instances selected. Instances that have the deployed App/Layer assigned will go through the process above, other instances will run a generic deploy phase (which allows those Instances to update their configuration in parallel). This process creates a good chance that all your ZEO clients will be restarted at once, causing a temporary outage and a slow site. If you

[159] This configuration assumes that the tarball has top-level directory called `eggs`. If you've setup such a tarball in an S3 bucket (usually creating it from your first instance deploy), you simply add this configuration to both the `deploy["plone_instances"]` and `deploy["zeoserver"]` Custom JSON before launching an instance.

have multiple Instances running ZEO clients you can deploy to them one at a time, to avoid an outage. You can also configure your Stack to do rolling deploys by adding a `restart_delay` in seconds to your Custom JSON under the top-level `plone_instances` key. When that is set, the deploy will wait that amount of time between each ZEO client restart for a given Instance.

If you use Travis CI to provide automatic testing of your buildout/application, you can configure Travis to automatically lanuch an Opsworks deploy for a specific Stack and Application on successful builds (see https://docs.travis-ci.com/user/deployment/opsworks/).

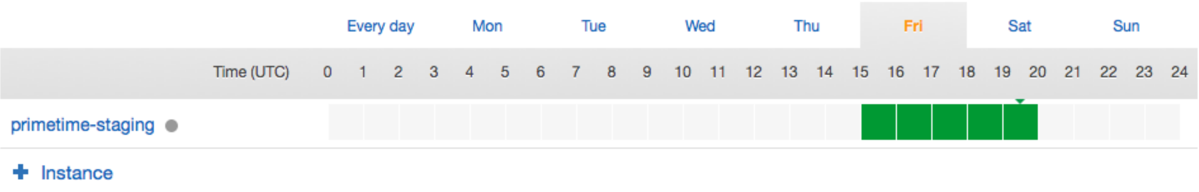
Instance Sizes

There are many available Instance types on EC2, which makes choosing the correct Instance sizes for your application cluster a bit of an art. The Opsworks recipes will automatically factor in the CPU capacity of the Instances you choose for your ZEO client Layers (using their Elastic Compute Unit/Core counts) to determine automatically how many ZEO clients to create per Instance. You can fine tune that calculation further by setting the `per_cpu` attribute under the `plone_instances` key in the Stack Custom JSON. You can also tweak the `zodb_cache_size`, and `zserver_threads` to help tune RAM usage for your ZEO clients.

Scaling

If you've setup a distributed blob storage (whether with NFS/GlusterFS, S3FS, ZEO or Relstorage), adding more ZEO clients is a simple matter of defining a new instance assigned only to your Plone Instances application Layer and starting it.

In addition to the normal 24/7 instances, you can define time-based Instances that automatically add instances during regular peak traffic periods.

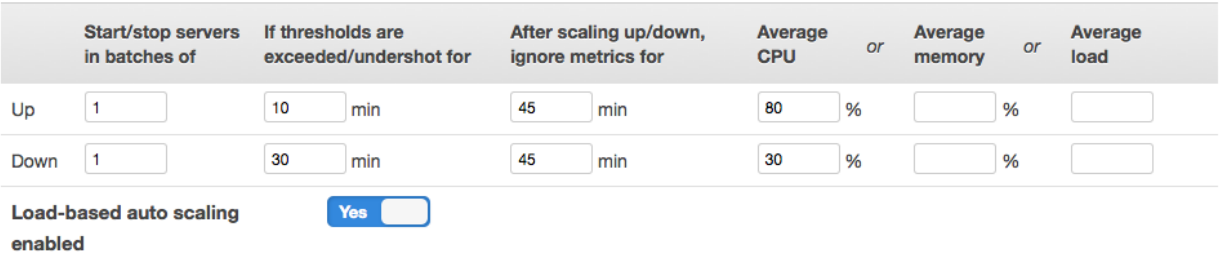


Every day Mon Tue Wed Thu Fri Sat Sun

Time (UTC) 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24

primetime-staging ● 🟩🟩🟩🟩🟩 (15–20)

Alternatively, you can define load-based instances which automatically start up and shutdown based on the average CPU usage, Load, or RAM usage of existing instances in the layer.



	Start/stop servers in batches of	If thresholds are exceeded/undershot for	After scaling up/down, ignore metrics for	Average CPU		Average memory		Average load
Up	1	10 min	45 min	80 %	or	%	or	
Down	1	30 min	45 min	30 %		%		

Load-based auto scaling enabled: Yes

Any new instances will automatically discover your existing ZEO server. Any load balancers will automatically discover any new ZEO clients. The Stack reconfiguration will happen automatically whenever an instance goes up or down.

You can view the HAProxy ZEO client status by visiting the password protected url `/balancer/stats` for your front end instance IP(s).

For a high traffic site that requires a high availability configuration, it may also make sense to run the front end HAProxy layer on multiple Instances in different Availablilty Zones. You would need to route external traffic to those servers using an adaptive DNS service or Amazon's Elastic Load Balancer.

Configuration

The Stack Custom JSON configuration offers a number of entry points for customizing the default Stack without needing to learn any Chef or Ruby. Those configuration parameters are thoroughly documented in the Plone Buildout cookbook README, and the full list of Plone Buildout cookbook specific attributes is in attributes/default.rb. Any of those attributes can be customized via the Stack Custom JSON. For example, the `nginx_plone["additional_configuration"]` and `nginx_plone["additional_servers"]` may be the most generically useful items for front end configuration.

The recipes and example buildout also include optional support for running and configuring a Solr search server and setting up a Celery task queue for running asynchronous jobs using collective.celery.

Maintenance

Backups

The recipes automatically setup weekly ZODB packing and log rotation. I like to Amazon's EBS snapshot feature for backups, and the EBS Snapshotting layer provides that functionality automatically.

It requires you to use the AWS IAM Console to create a new user with the following permissions:

```
ec2:CreateSnapshot
ec2:CreateTags
ec2:DeleteSnapshot
ec2:DescribeInstances
ec2:DescribeSnapshots
```

You will need to note the API credentials for this new user and enter them into the Stack Custom JSON as follows:

```
"ebs_snapshots" : {
    "aws_key" : "***** AWS KEY FOR SNAPSHOTTING (IAM USER) *****",
    "aws_secret" : "***** AWS SECRET FOR SNAPSHOTTING (IAM USER) *****"}
}
```

The EBS Snapshotting Layer should be assigned to any production instance which has EBS volumes on which you are storing data. Generally speaking, any production instance with the Zeoserver, Shared Blob, or Solr Layers assigned should also have the EBS Snapshotting Layer assigned.

This Layer will setup automatic nightly snapshots of all mounted EBS volumes. By default it retains up to 15 snapshots, but that can be configured setting `ebs_snapshots["keep"]` to the number you wish to retain in the Stack Custom JSON.

Updates

Ubuntu security and OS package updates can be automated by adding the following Custom JSON config:

```
"apt": {
    "unattended_upgrades": {
      "package_blacklist": [],
      "enable": true,
        "mail": "youremail@here",
        "auto_fix_interrupted_dpkg": true
    }
}
```

Monitoring

AWS provides various monitoring and alerting features, but most alerting features need to be manually configured on a per EC2 instance basis. That's not so convenient for a stack of instances which may grow, shrink or change over time. For that reason I like to use New Relic for server monitoring. There is built-in integration in the recipes, which includes detailed performance server and client performance monitoring for Plone, as well as plugins for Nginx, Varnish and HAProxy services and standard CPU, Disk space and RAM server metrics.

There's also a recipe provided to integrate the Papertrail log tracing and searching service. To help you live the dream of never having to SSH into your servers.

Sending Mail

It's possible, and not difficult to install and configure a mailer using a chef postfix recipe and some more Custom JSON. However, I do not recommend doing so. Cloud Servers generally, and EC2 specifically tend to land on SPAM blacklists, ensuring your outgoing mail is not blackholed generally requires some special care and requests to Amazon to setup reverse DNS and whitelist any outgoing mail servers.

Instead I recommend using a hosted mail delivery service like Amazon SES or perhaps GMail.

SSH Access

Ideally, you never have to login to your cloud server, but things go wrong and you might have to eventually, even if only out of curiosity. By default OpsWorks does not assign an SSH key to new instances, but you can set one if desired at either the instance or the Stack level. Better yet, Opsworks allows more granular access control in combination with IAM. If you create a user via the AWS IAM console (no permissions need be assigned, and no credentials added or recorded for SSH access), you can then import that user into the OpsWorks Users control panel. In OpsWorks users can be given access to specific stacks, allowing them to view, deploy or manage them, as well as granting them SSH and/or sudo accees to Stack Instances using a public key that can be added through the web interface. Once you've imported an IAM user into Opsworks and granted it SSH access with a public key, that user should be able to log in to all instances in the stack. *[0]

Note: A note on OS permissions: all application related files live under `/srv/www` and are generally owned by the `deploy` user with fairly restricted permissions. Any user SSH'ing in will probably need to sudo to the `deploy` user to see or do much of interest.

What Doesn't It Do

Storage Options

Amazon recently introduced `Elastic File System` an effectively unlimited size cloud file storage that can be mounted simultaneously on multiple servers. It provides high availability and durability and should be significantly

[0] You should *never* manually modify any configuration on a cloud configured server, except for purposes of troubleshooting. Any changes you make to the server should be made via the Stack configuration (i.e. the Custom JSON and the Recipes assigned to Layers).

faster than either S3 or even standard SSD EBS mounts. For these reasons it would make an ideal storage option for a shared blob directory and possibly also ZEO filestorages.

Integrating this new storage option into the recipes and documentation should be a high priority going forward. The interface for Elastic File System is NFS v4, which the stack already supports, so it may even be trivial to integrate.

There are probably some other fun new AWS services that would be useful to integrate.

Proxy Cache Purging

Plone provides some very nice proxy caching configuration, but that configuration is managed TTW and stored persistently in the ZODB. If you have multiple proxy caches which could be going online or offline automatically or changing IP addresses, then having persistent configuration of caches to purge is not ideal.

It would be very useful to add support in plone.app.caching for reading a list of proxy servers from an environment variable or other mechanism that can easily be managed as part of the configuration phase.

Chef 12

The latest Opsworks codebase requires Chef 12. The Python cookbooks are currently only tested on Chef 11. Running Opsworks on Ubuntu Xenial instances requires using the latest Chef 12 version. This will likely require extensive testing and upgrades to dependency cookbooks.

Other Stuff?

Probably, play around with it and let me know.

“Through-the-web” Plone customization

Warning: This chapter is still work in progress!

Basic: Customizing logo and CSS of default theme

In this section you will:

- Use the Site control panel to add a custom logo
- Customize the look of a Plone site by adjusting Less Variables
- Add a custom toolbar logo

Topics covered:

- The “Site” control panel
- The “Resource Registries” Control Panel
- Resource Registries > Development Mode

Customize logo

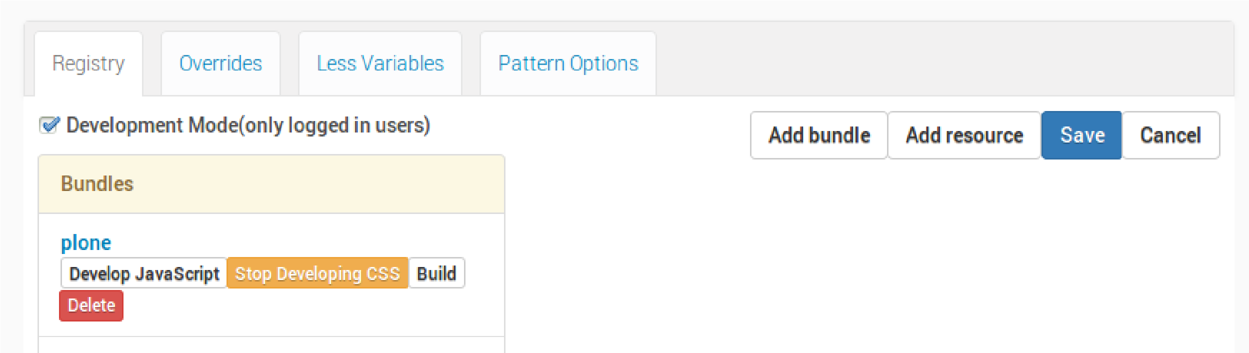
1. Go to the Plone Control Panel: *toolbar* → *admin* → *Site Setup*
2. Go to the "Site" control panel.
3. You will see this form:
4. You can now add / remove your custom logo

See the official docs.

Customize CSS/Less variables

1. Go back to the Control Panel.
2. Go to the *Resource Registries* control panel.
3. On the first tab: enable *Development Mode*.
4. In the "plone" bundle below, click on "develop CSS".

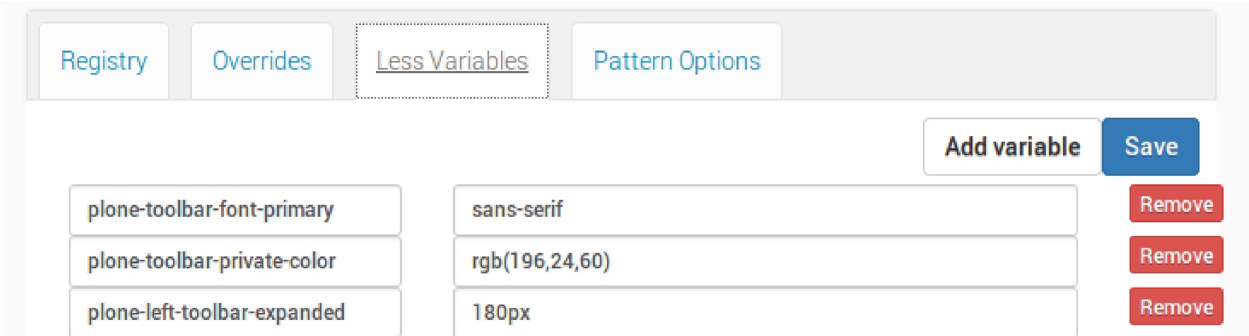
Your panel should now look like this:



☑ Development Mode(only logged in users)

Now we can play with some Less variables:

1. Go to the *Less Variables* tab.
2. Find the variable `plone-left-toolbar-expanded` and set it to 400px.



plone-toolbar-font-primary	sans-serif
plone-toolbar-private-color	rgb(196,24,60)
plone-left-toolbar-expanded	180px

3. Hit the *Save* button in the upper right and reload the page.
4. Click on the toolbar logo to expand the toolbar: voilá!

You can play around with some other variables, if you want.

Warning: "Development Mode" is really expensive for the browser. Depending on the browser and on the system, you might encounter extreme slowness while rendering the page. You could see an unthemed page for a while. Remember to switch it off as soon as you finished tweaking.

Configuring and Customizing Plone "Through The Web"

Warning:

This chapter has not yet been updated for Plone 5!

The Control Panel

The most important parts of Plone can be configured in the control panel.

- Click on the portrait/username in the toolbar
- Click *Site Setup*

We'll explain every page and mention some of the actions you can perform here.

General

1. Date and Time
2. Language
3. Mail
4. Navigation
5. Site
6. Add-ons
7. Search
8. Discussion
9. Theming
10. Social Media
11. Syndication
12. TinyMCE

Content

1. Content Rules
2. Editing
3. Image Handling
4. Markup

5. Content Settings
6. Dexterity Content Types

Users

1. Users and Groups

Security

1. HTML Filtering
2. Security
3. Errors

Advanced

1. Maintenance
2. Management Interface
3. Caching
4. Configuration Registry
5. Resource Registries

Below the links you will find information on your Plone, Zope and Python Versions and an indicator as to whether you're running in production or development mode.

Change the logo

Let's change the logo.

- Download a ploneconf logo: https://www.starzel.de/plone-tutorial/ploneconf-logo-2014
- Go to http://localhost:8080/Plone/@@site-controlpanel
- Upload the Logo.

See also:

http://docs.plone.org/adapt-and-extend/change-the-logo.html

Portlets

In the toolbar under *More options* you can open the configuration for the different places where you can have portlets.

- UI fit for smart content editors
- Various types
- Portlet configuration is inherited
- Managing
- Ordering/weighting

- The future: may be replaced by tiles
- `@@manage-portlets`

Example:

- Go to http://localhost:8080/Plone/@@manage-portlets
- Add a static portlet "Sponsors" on the right side.
- Remove the news portlet and add a new one on the left side.
- Go to the training folder: http://localhost:8080/Plone/the-event/training and click `Manage portlets`
- Add a static portlet. "Featured training: Become a Plone-Rockstar at Mastering Plone!"
- Use the toolbar to configure the portlets of the footer:
  - Hide the portlets "Footer" and "Colophon".
  - Add a "Static text portlet" enter "Copyright 2015 by Plone Community".
  - Use "Insert > Special Character" to add a real © sign.
  - You could turn that into a link to a copyright page later.

Viewlets

Portlets save data, Viewlets usually don't. Viewlets are often used for UI-Elements and have no nice UI to customize them.

- `@@manage-viewlets`
- Viewlets have no nice UI
- Not aimed at content editors
- Not locally addable, no configurable inheritance.
- Usually global (depends on code)
- Will be replaced by tiles?
- The code is much simpler (we'll create one tomorrow).
- Live in viewlet managers, can be nested (by adding a viewlet that contains a viewlet manager).
- TTW reordering only within the same viewlet manager.
- The code decides when it is shown and what it shows.

ZMI (Zope Management Interface)

Go to http://localhost:8080/Plone/manage

Zope is the foundation of Plone. Here you can access the inner workings of Zope and Plone alike.

Note: Here you can easily break your site so you should know what you are doing!

We only cover three parts of customization in the ZMI now. Later on when we added our own code we'll come back to the ZMI and will look for it.

At some point you'll have to learn what all those objects are about. But not today.

Actions (portal_actions)

- Actions are mostly links. But **really flexible** links.
- Actions are configurable ttw and through code.
- These actions are usually iterated over in viewlets and displayed.

Examples:

- Links in the Footer (`site_actions`)
- Actions Dropdown (`folder_buttons`)

Actions have properties like:

- description
- url
- i18n-domain
- condition
- permissions

`site_actions`

These are the links at the bottom of the page:

- Site Map
- Accessibility
- Contact
- Site Setup

We want a new link to legal information, called "Imprint".

- Go to `site_actions` (we know that because we checked in `@@manage-viewlets`)
- Add a CMF Action `imprint`
- Set URL to `string:${portal_url}/imprint`
- Leave *condition* empty
- Set permission to `View`
- Save

explain

- Check if the link is on the page
- Create new Document *Imprint* and publish

See also:

http://docs.plone.org/develop/plone/functionality/actions.html

Global navigation

- The horizontal navigation is called `portal_tabs`
- Go to *portal_actions* → *portal_tabs* Link
- Edit `index_html`

Where is the navigation?

The navigation shows content-objects, which are in Plone's root. Plus all actions in `portal_tabs`.

Explain & edit `index_html`

Configuring the navigation itself is done elsewhere: http://localhost:8080/Plone/@@navigation-controlpanel

If time explain:

- user > undo (cool!)
- user > login/logout

Skins (`portal_skins`)

In `portal_skins` we can change certain images, CSS-files and templates.

- `portal_skins` is deprecated technology
- Plone 5 got rid of most files that lived in `portal_skins`.

Change some CSS

- Go to ZMI
- Go to `portal_skins`
- Go to `plone_styles`
- Go to `ploneCustom.css`
- Click *customize*

The CSS you add to this file is instantly active on the site.

portal_view_customizations

Change the footer

- Go to `portal_view_customizations`
- Search `plone.footer`, click and customize
- Replace the content with the following

```
<div i18n:domain="plone"
     id="portal-footer">
   <p>&copy; 2016 by me! |
     <a href="mailto:info@ploneconf.org">
      Contact us
     </a>
```

```
    </p>
</div>
```

See also:

http://docs.plone.org/adapt-and-extend/theming/templates_css/skin_layers.html

CSS Registry (`portal_css`)

deprecated (See the chapter on theming)

Further tools in the ZMI

There are many more notable items in the ZMI. We'll visit some of them later.

- *acl_users*
- *error_log*
- *portal_properties* (deprecated)
- *portal_setup*
- *portal_workflow*
- *portal_catalog*

Summary

You can configure and customize a lot in Plone through the web. The most important options are accessible in the Plone control panel but some are hidden away in the ZMI. The amount and presentation of information is overwhelming but you'll get the hang of it through a lot of practice.

TTW Theming I: Introduction to Diazo Theming

In this section you will:

- Use the "Theming" control panel to make a copy of Plone's default theme (barceloneta)
- Customize a theme using Diazo rules
- Customize a theme by editing and compiling Less files

Topics covered:

- Diazo and plone.app.theming
- "Barceloneta" - The Default Plone Theme
- The "Theming tool"
- Building CSS in the "Theming tool"
- `<body>` element CSS classes
- Conditionally activating rules

Installation

We will use a Plone pre-configured Heroku instance.

Once deployed, create a Plone site.

Two approaches to theming

There are two main approaches to creating a custom theme:

1. Copying the default Barceloneta theme
2. Inheriting from the default Barceloneta theme.

In this section we'll look at the first approach, part II will explore the second approach.

What is Diazo?

`Diazo` is a theming engine used by Plone to make theming a site easier. At its core, a Diazo theme consists of an HTML page and `rules.xml` file containing directives.

Note: You can find extended information about Diazo and its integration package `plone.app.theming` in the official docs: Diazo docs and plone.app.theming docs.

Principles

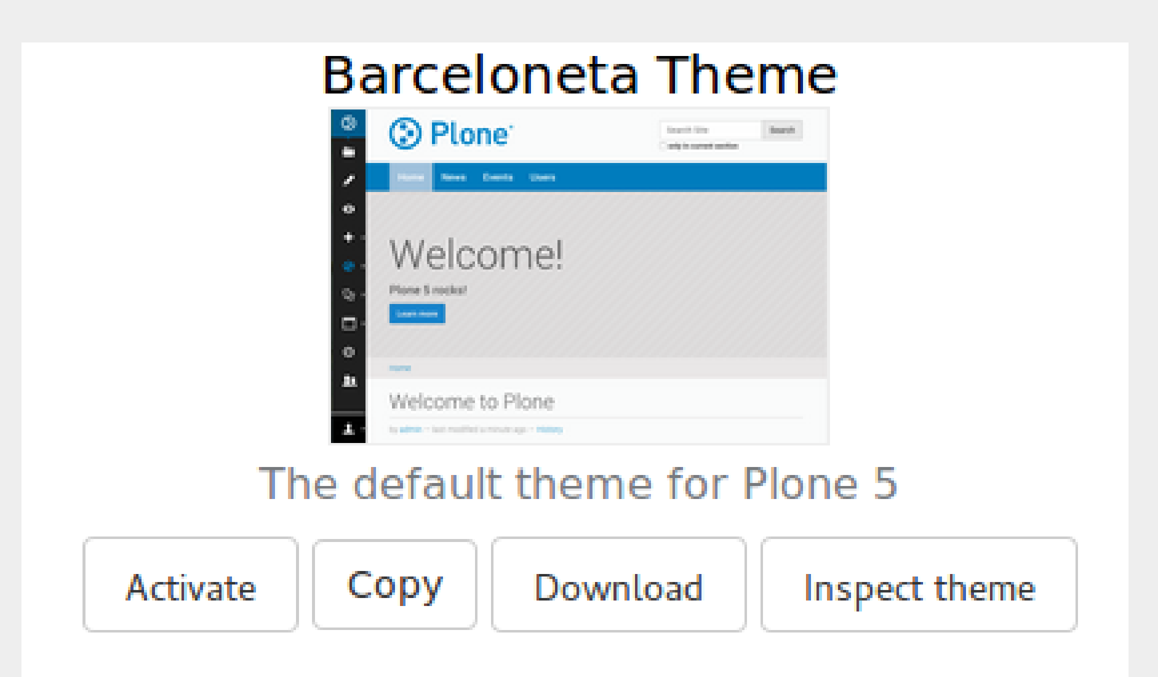
For this part of the training you just need to know the basic principles of a Diazo theme:

- Plone renders the content of the page;
- Diazo rules inject the content into any static theme;

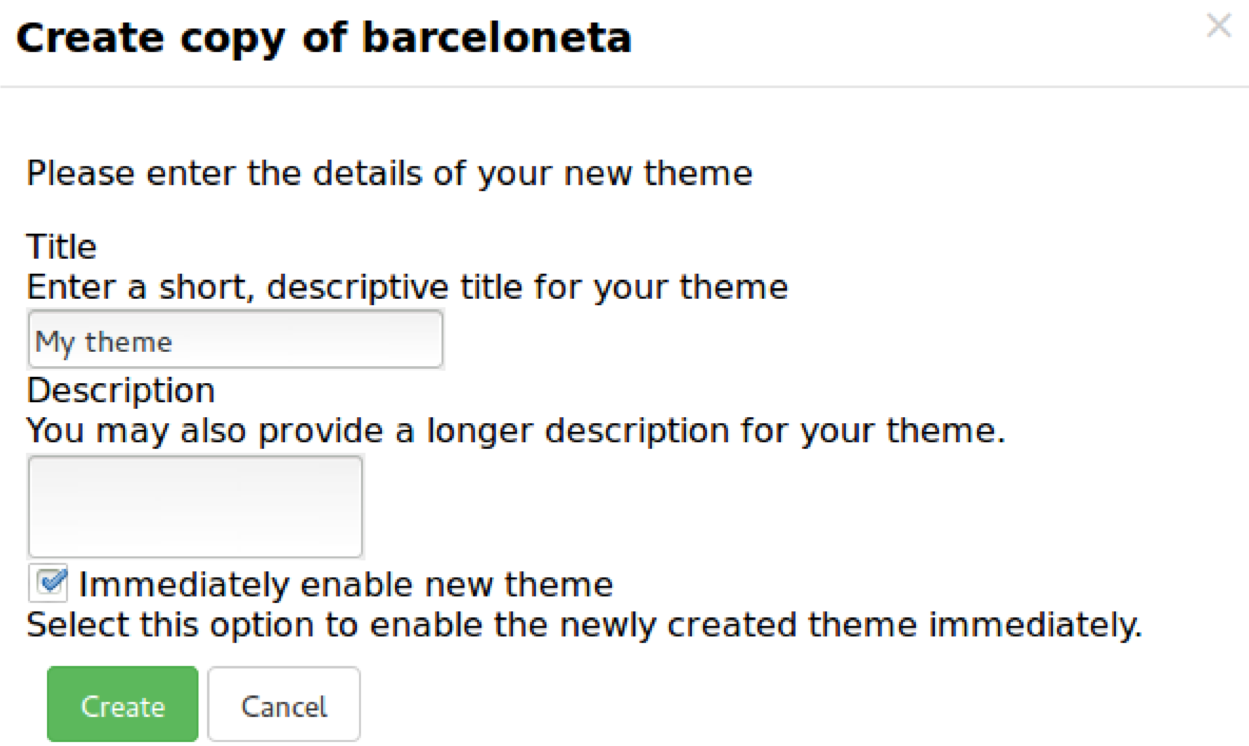
Copy barceloneta theme

To create our playground we will copy the existing Barceloneta theme.

1. go to the *Theming* control panel
2. you will see the available themes. In a bare new Plone site, you will see something like this:



3. click on the *Copy* button and get to the copy form
4. insert "My theme" as the name and activate it by default



Create copy of barceloneta

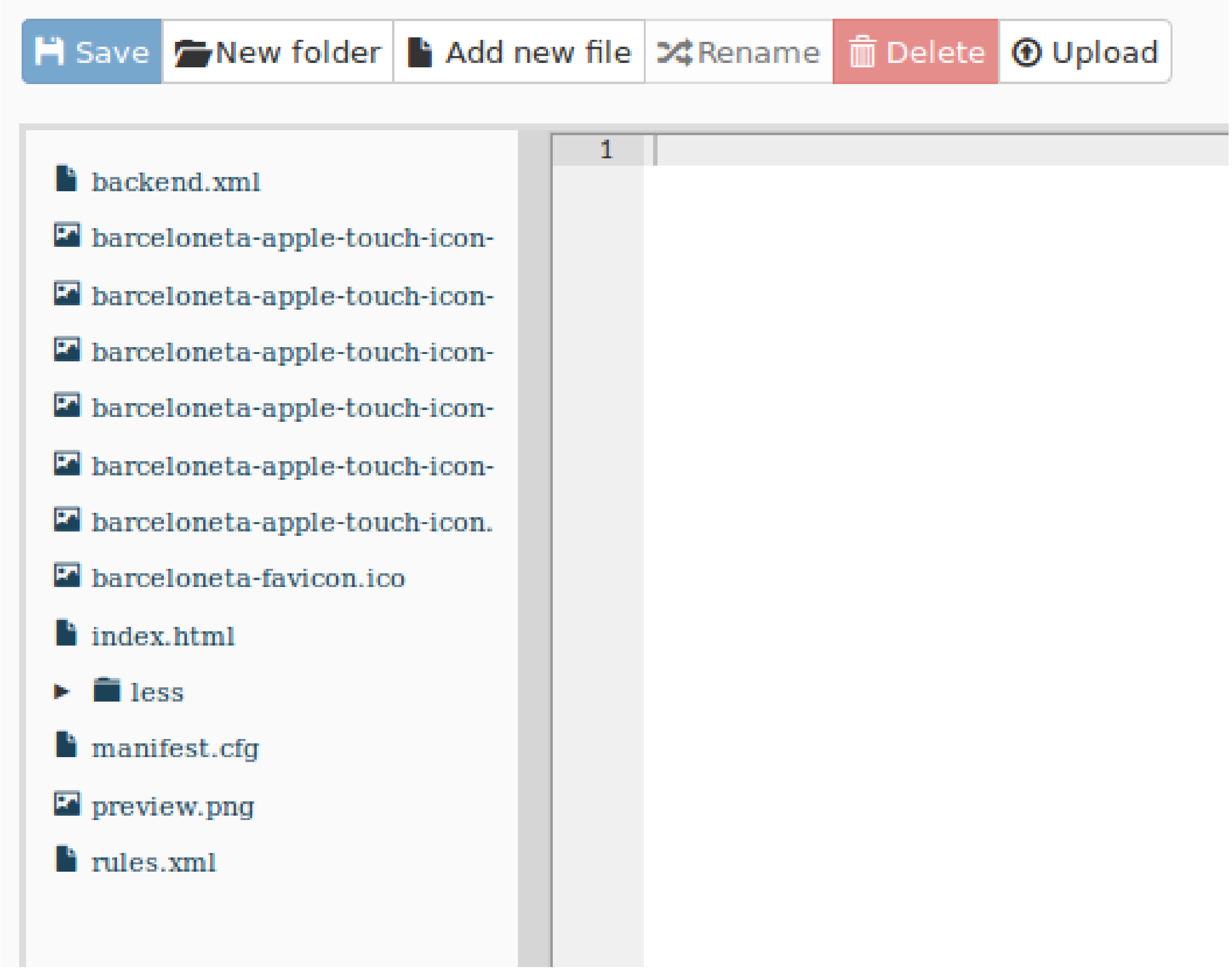
Please enter the details of your new theme

Title
Enter a short, descriptive title for your theme
My theme

Description
You may also provide a longer description for your theme.

☑ Immediately enable new theme
Select this option to enable the newly created theme immediately.

5. click on *Create* and you get redirected to your new theme's inspector:


Save
New folder
Add new file
Rename
Delete
Upload
backend.xml
barceloneta-apple-touch-icon-
barceloneta-apple-touch-icon-
barceloneta-apple-touch-icon-
barceloneta-apple-touch-icon-
barceloneta-apple-touch-icon-
barceloneta-apple-touch-icon.
barceloneta-favicon.ico
index.html
less
manifest.cfg
preview.png
rules.xml
1

Anatomy of a Diazo theme

The most important files:

- `manifest.cfg`: contains metadata about the theme (manifest reference);
- `rules.xml`: contains the theme rules (rules reference);
- `index.html`: the static HTML of the theme.

Exercise 1 - Inspecting the `manifest.cfg`

To better understand how your theme is arranged start by reading the `manifest.cfg` file.

In the theming tool, open the `manifest.cfg` spend a minute or two looking through it, then see if you can answer the questions below.

Where are the main rules located for your theme?

What property in the `manifest.cfg` file defines the source CSS/Less file used by the theme?

What do you think is the purpose of the `prefix` property?

Solution

The main rules are defined by the `rules` property (you could point this anywhere, however the accepted convention is to use a file named `rules.xml`.

The `development-css` property points at the main Less file, when compiled to CSS it is placed in the location defined by the `production-css` property.

The `prefix` property defines the default location to look for non prefixed files, for example if your prefix is set to `/++theme++mytheme` then a file like index.html will be expected at `/++theme++mytheme/index.html`

<body> CSS classes

As you browse a Plone site, Plone adds rich information about your current context. This information is represented as special classes in the `<body>` element. Information represented by the `<body>` classes includes:

- the current user role, and permissions,
- the current content-type and its template,
- the site section and sub section,
- the current subsite (if any),
- whether this is a frontend view,
- whether icons are enabled.

<body> classes for an anonymous visitor

Below you can see an example of the body classes for a page named "front-page", located in the root of a typical Plone site called "acme":

```
<body class="template-document_view
             portaltype-document
             site-acme
             section-front-page
             icons-on
             thumbs-on
             frontend
             viewpermission-view
             userrole-anonymous">
```

<body> classes for a manager

And here is what the classes for the same page look like when viewed by a manager that has logged in:

```
<body class="template-document_view
             portaltype-document
             site-acme
             section-front-page
             icons-on
             thumbs-on
             frontend
             viewpermission-view
             userrole-member
             userrole-manager
             userrole-authenticated
             plone-toolbar-left
             plone-toolbar-expanded
             plone-toolbar-left-expanded">
```

Notice the addition of `userrole-manager`.

Exercise 2 - Discussion about the `<body>` classes

Look back at the `<body>` classes for a manager then see if you can answer the following questions.

1. What other roles does the manager have?
2. Can you see other differences?
3. What do you think the `plone-toolbar-expanded` class does?

Solution

The manager also has the role "member" and "authenticated"

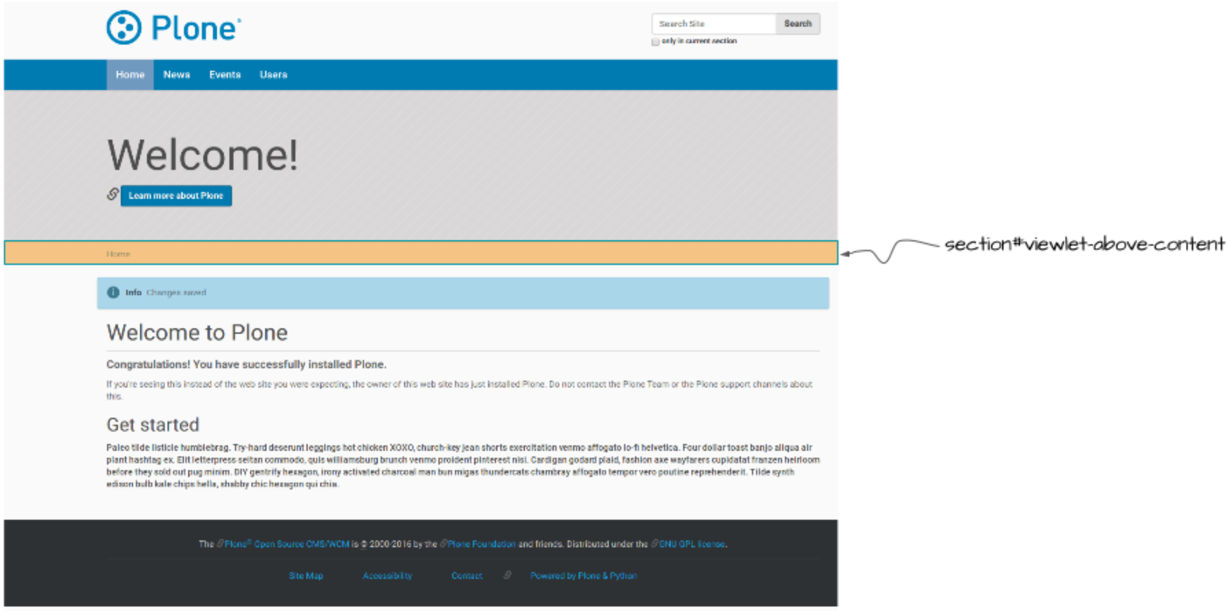
There are `plone-toolbar` classes added to the `<body>` element, these control the display of the toolbar

The `plone-toolbar-expanded` class is used to control styles used by the expanded version of the toolbar.

Custom rules

Let's open `rules.xml`. You will see all the rules that are used in the Barceloneta theme right now. For the time being let's concentrate on how to hack these rules.

Conditionally showing content



Welcome!

Changes saved

Welcome to Plone

Congratulations! You have successfully installed Plone.

If you're seeing this instead of the web site you were expecting, the owner of this web site has just installed Plone. Do not contact the Plone Team or the Plone support channels about this.

Get started

Paleo tilde listicle humblebrag. Try-hard deserunt leggings hot chicken XOXO, church-key jean shorts exercitation venmo affogato lo-fi helvetica. Four dollar toast banjo aliqua air plant hashtag ex. Elit letterpress seitan commodo, quis williamsburg brunch venmo proident pinterest nisi. Cardigan godard plaid, fashion axe wayfarers cupidatat franzen heirloom before they sold out pug minim. DIY gentrify hexagon, irony activated charcoal man bun migas thundercats chambray affogato tempor vero poutine reprehenderit. Tilde synth edison bulb kale chips hella, shabby chic hexagon qui chia.

The Plone® Open Source CMS/WCM is © 2000-2016 by the Plone Foundation and friends. Distributed under the GNU GPL license.

Suppose that we want to make the "above content" block (the one that contains breadcrumbs) conditional, and show it only for authenticated users.

In the `rules.xml` find this line:

```
<replace css:content="#viewlet-above-content" css:theme="#above-content" />
```

This rule states that the element that comes from the content (Plone) with the id `#viewlet-above-content` must replace the element with the id `#above-content` in the static theme.

We want to hide it for anonymous users (hint: We'll use the `<body>` classses discussed above).

The class we are looking for is `userrole-authenticated`. Add another property to the rule so that we produce this code:

```
<replace
    css:if-content="body.userrole-authenticated"
    css:content="#viewlet-above-content"
    css:theme="#above-content" />
```

The attribute `css:if-content` allows us to put a condition on the rules based on a CSS selector that acts on the content. In this way the rule will be applied only if the body element has the class `.userrole-authenticated`.

We will learn more about Diazo rules in *TTW Theming II: Creating a custom theme based on Barceloneta*.

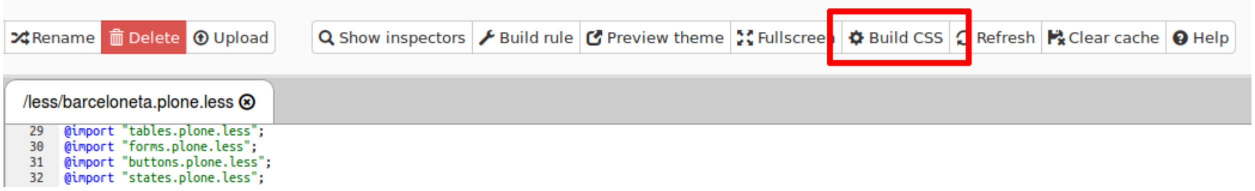
Customize CSS

1. from theming tool open the file `less/barceloneta.plone.less`, that is the main Less file as specified in the manifest;
2. add your own customization at the bottom, like:

```
body{ background-color: red; font-size: 18px ;};
```

Note: Normally you would place this in a separate file to keep the main one clean but for this example it is enough.

3. push the buttons *Save* and *Build CSS*



```
@import "tables.plone.less";
@import "forms.plone.less";
@import "buttons.plone.less";
@import "states.plone.less";
```

4. go back to the Plone site and reload the page: voilá!

Warning: At the moment you need to "Build CSS" from the main file, the one declared in the manifest (in this case `less/barceloneta.plone.less`). So, whatever Less file you edit, go back to the main one to compile. This behavior will be improved but for now, just remember this simple rule ;)

TTW Theming II: Creating a custom theme based on Barceloneta

In this section you will:

- Create a theme by inheriting from the Barceloneta theme.
- Using the `manifest.cfg`, register a production CSS file.
- Use an XInclude to incorporate rules from the Barceloneta theme.
- Use `?diazo.off=1` to view unthemed versions.
- Use conditional rules to have a different backend theme from the anonymous visitors theme.

Topics covered:

- Inheriting from Barceloneta.
- Diazo rule directives and attributes.
- Viewing the unthemed version of a Plone item.
- Creating a visitor-only theme.

Inheriting from Barceloneta

Key Ideas

When inheriting from the Barceloneta theme keep the following in mind:

- The theme provides styles and assets used by Plone's backend tools.
- Inheritance involves including the Barceloneta `rules.xml` (`++theme++barceloneta/rules.xml`) and styles.
- The prefix/unique path to the Barceloneta theme is `++theme++barceloneta`.
- It is necessary to include a copy of Barceloneta's `index.html` in the root of your custom theme.

- The three key files involved are `manifest.cfg`, `rules.xml` and a Less file defined in the manifest which we will call `styles.less`.
- Use "Build CSS" to generate a CSS file from your custom Less file.

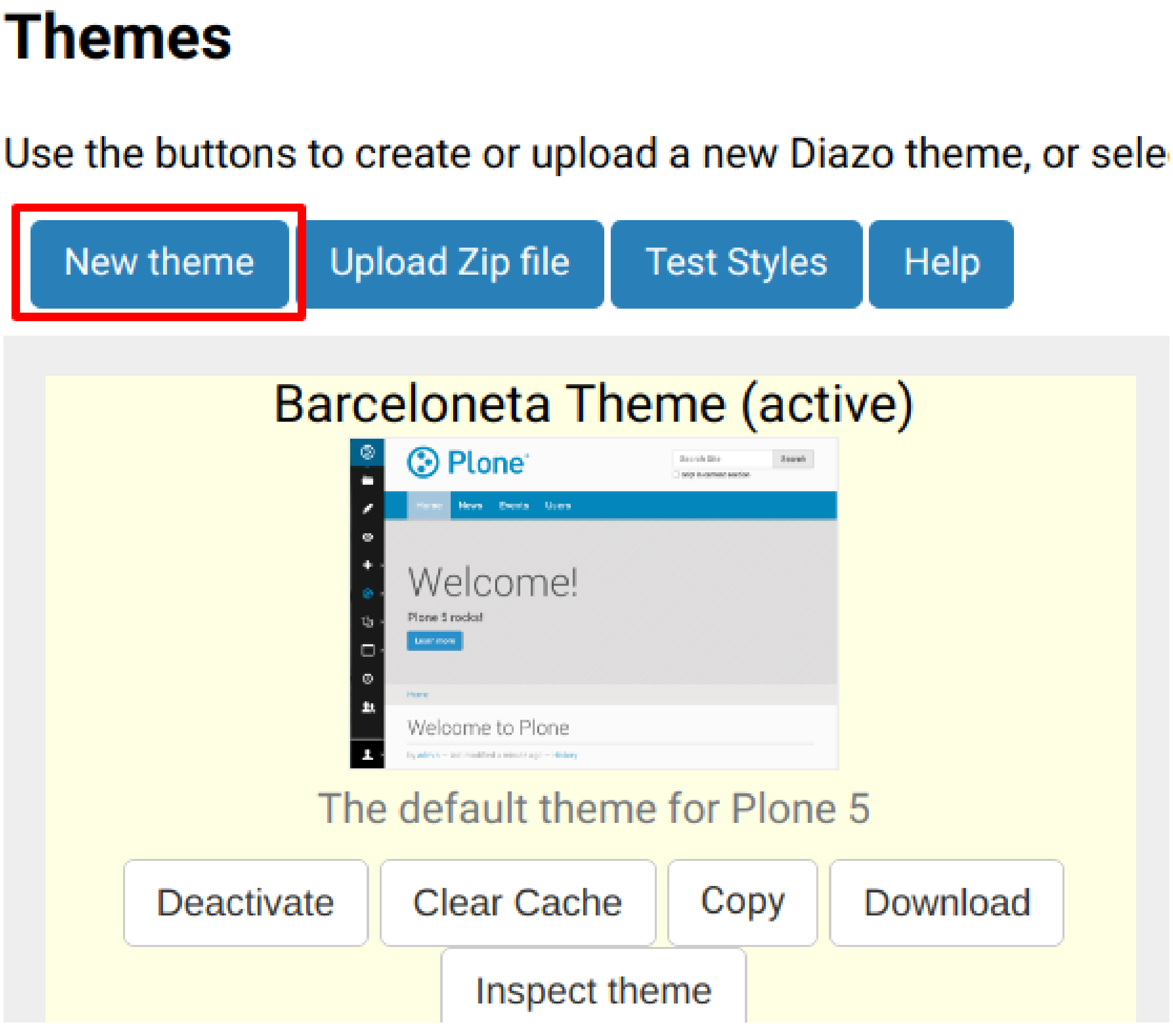
Copying Barceloneta makes your theme heavier and will likely make upgrading more difficult.

The Barceloneta theme provides many assets used by Plone's utilities that you do not need to duplicate. Additionally new releases of the theme may introduce optimizations or bug fixes. By referencing the Barceloneta rules and styles, instead of copying them, you automatically benefit from any updates to the Barceloneta theme while also keeping your custom theme relatively small.

Exercise 1 - Create a new theme that inherits from Barceloneta

In this exercise we will create a new theme that inherits the Barceloneta rules and styles.

1. Create a new theme



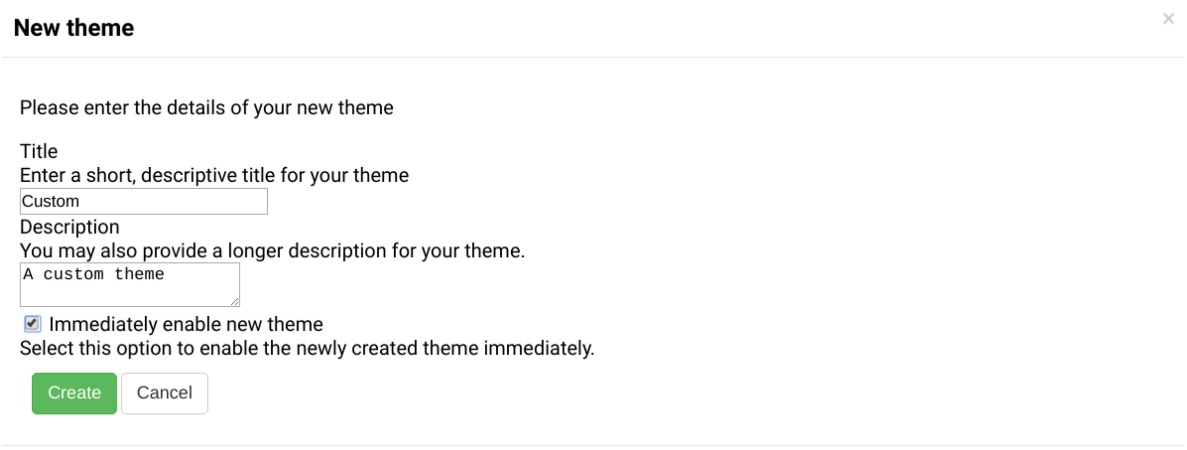
Themes

Use the buttons to create or upload a new Diazo theme, or sele

Barceloneta Theme (active)

The default theme for Plone 5

and name it "Custom"



New theme

Please enter the details of your new theme

Title
Enter a short, descriptive title for your theme
Custom

Description
You may also provide a longer description for your theme.
A custom theme

☑ Immediately enable new theme
Select this option to enable the newly created theme immediately.

2. In the theming editor, ensure that your new theme contains the files `manifest.cfg`, `rules.xml`, `index.html` (from Barceloneta) and `styles.less`.

- `manifest.cfg`, declaring your theme:

```
[theme]
title = mytheme
description =
development-css = ++theme++custom/styles.less
production-css = ++theme++custom/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 generate the `styles.css` file using `styles.less` and the "Build CSS" button.

Your theme is ready.

Diazo rule directives and attributes

The Diazo rules file is an XML document containing rules to specify where the content elements (title, footer, main text, etc.) will be located in the targeted theme page. The rules are created using *rule directives* which have *attributes*; attribute values are either CSS expressions or XPath expressions.

CSS selector based attributes

It is generally recommended that you use CSS3 selectors to target elements in your content or theme. The CSS3 selectors used by Diazo directives are listed below:

`css:theme` Used to select target elements from the theme using CSS3 selectors.

`css:content` Used to specify the element that should be taken from the content.

`css:theme-children` Used to select the children of matching elements.

`css:content-children` Used to identify the children of an element that will be used.

XPath selector based attributes

Depending on complexity of the required selector it is sometimes necessary or more convenient to use XPath selectors instead of CSS selectors. XPath selectors use the unprefixed attributes `theme` and `content`. The common XPath selector attributes include:

`theme` Used to select target elements from the theme using XPath selectors.

`content` Used to specify the element that should be taken from the content using XPath selectors.

`theme-children` Used to select the children of matching elements using XPath selectors.

`content-children` Used to identify the children of an element that will be used using XPath selectors.

You can also create conditions about the current path using `if-path`.

Note: For a more comprehensive overview of all the Diazo rule directives and related attributes see: http://docs.diazo.org/en/latest/basic.html#rule-directives

Viewing the unthemed Plone site

When you create your Diazo rules, it is important to know how the content Diazo is receiving from Plone is structured. In order to see a "non-diazoed" version page, just add `?diazo.off=1` at the end of its URL.

Exercise 2 - Viewing the unthemed site

1. Use `diazo.off=1` to view the unthemed version of your site.
2. Using your browser's inspector, find out the location/name of some of Plone's elements. Then try to answer the following:

   What do you think is the difference between "content-core" and "content"? There are several viewlets, how many do you count? Can you identify any portlets, what do you think they are for?

   Solution

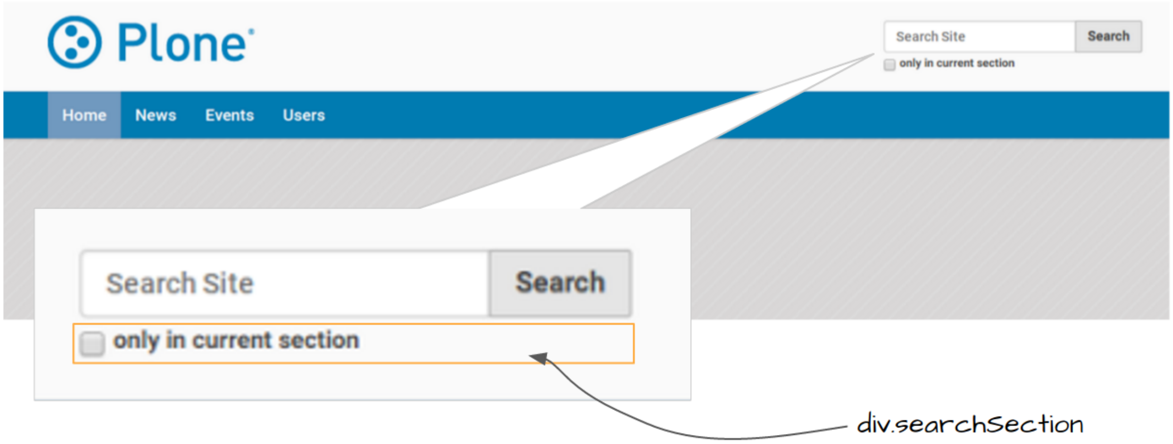
   The "content-core" does not include the "title" and "description" while the "content" combines the "title", "description" and "content-core".

   Out of the box there are six viewlets (`viewlet-above-content`, `viewlet-above-content-title` `viewlet-below-content-title`, `viewlet-above-content-body`, `viewlet-below-content-body`, `viewlet-below-content`).

   There are a few *footer* portlets which construct the footer of the site.

Exercise 3 - the `<drop>` directives

1. Add a rule that drops the "search section" checkbox from the search box. See the diagram below:



Conditional attributes

The following attributes can be used to conditionally activate a directive.

`css:if-content` Defines a CSS3 expression: if there is an element in the *content* that matches the expression then activate the directive.

`css:if-theme` Defines a CSS3 expression: if there is an element in the *theme* that matches the expression then activate the directive.

`if-content` Defines an XPath expression: if there is an element in the *content* that matches the expression then activate the directive.

if-theme Defines an XPath expression: if there is an element in the *theme* that matches the expression then activate the directive.

if-path Conditionally activate the current directive based on the current path.

Note: In a previous chapter we discussed the Plone `<body>` element and how to take advantage of the custom CSS classes associated with it. We were introduced to the attribute `css:if-content`. Remember that we are able to determine a lot of context related information from the classes, such as:

```
- the current user role, and its permissions,
- the current content-type and its template,
- the site section and sub section,
- the current subsite (if any).
```

Here is an example

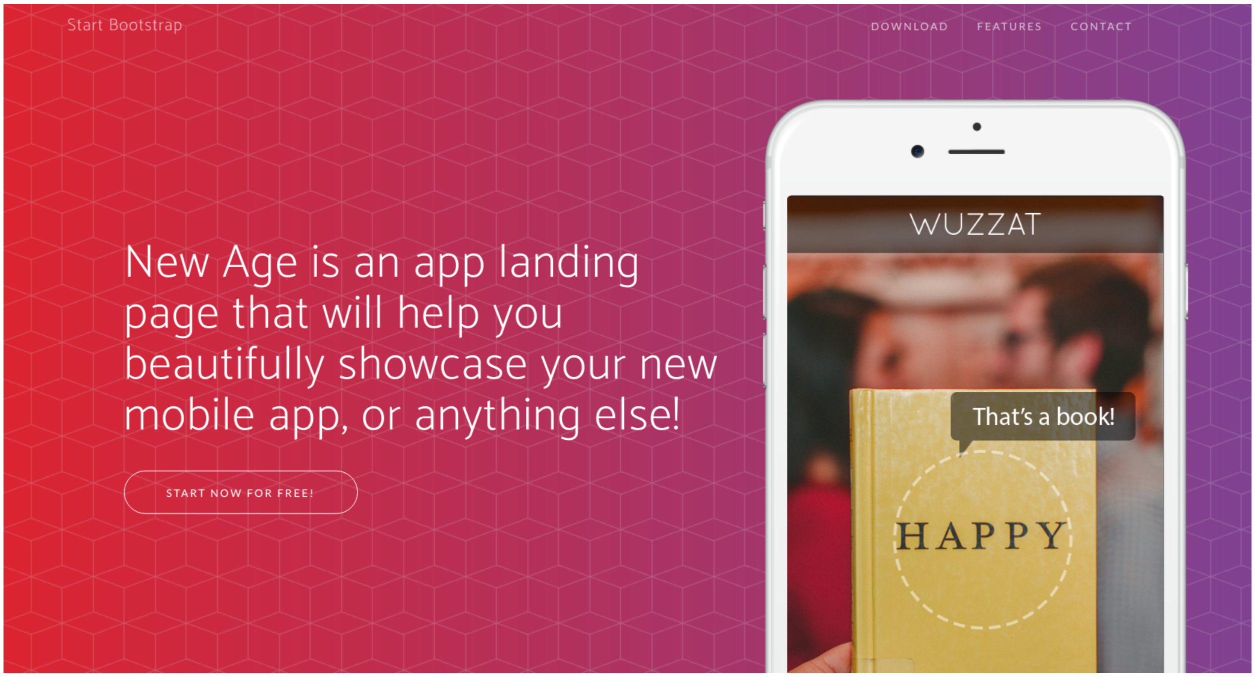
```
<body class="template-summary_view
             portaltype-collection
             site-Plone
             section-news
             subsection-aggregator
             icons-on
             thumbs-on
             frontend
             viewpermission-view
             userrole-manager
             userrole-authenticated
             userrole-owner
             plone-toolbar-left
             plone-toolbar-expanded
             plone-toolbar-left-expanded
             pat-plone
             patterns-loaded">
```

Converting an existing HTML template into an theme

In the Plone "universe" it is not uncommon to convert an existing HTML template into a Diazo theme. Just ensure that when you zip up the source theme that there is a single folder in the root of the zip file. We will explore this in more detail in the next exercise.

Exercise 4 - Convert a HTML template into a Diazo theme

In this exercise we will walk through the process of converting an existing free HTML theme into a Diazo-based Plone theme.

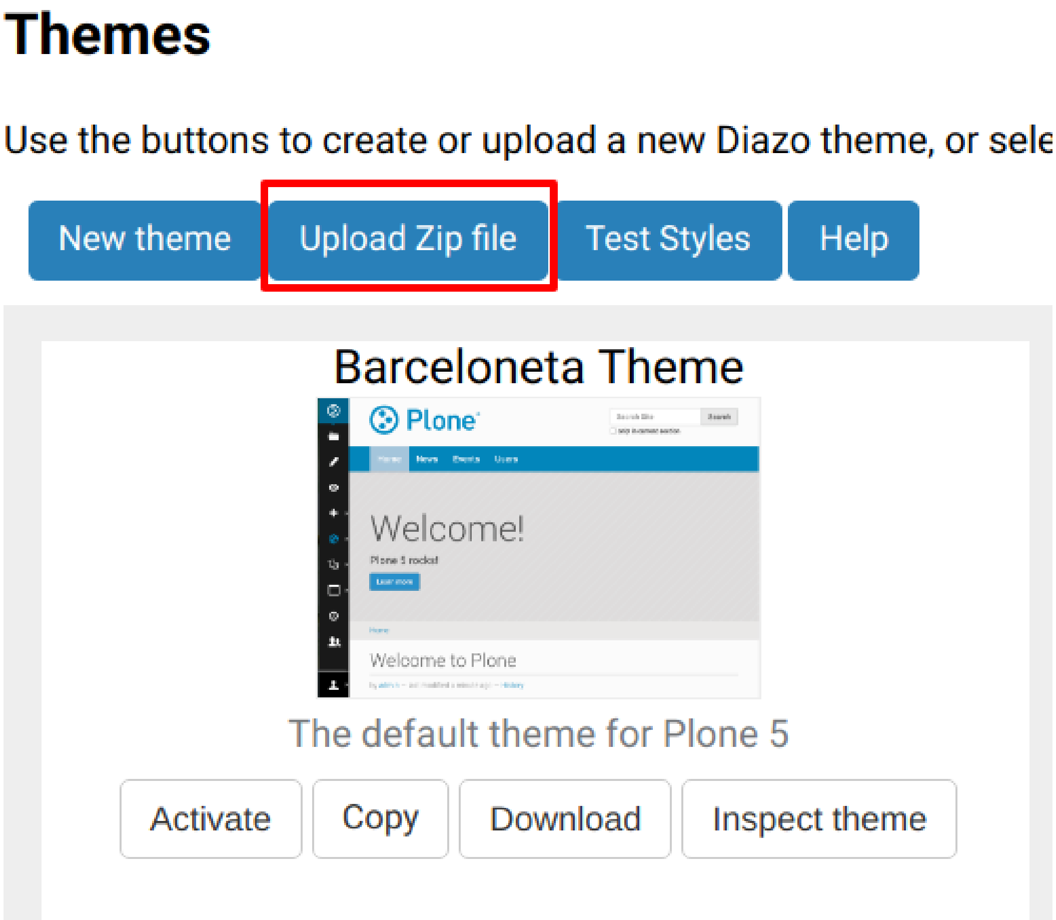

Start Bootstrap
DOWNLOAD
FEATURES
CONTACT
New Age is an app landing page that will help you beautifully showcase your new mobile app, or anything else!
START NOW FOR FREE!
WUZZAT
That's a book!
HAPPY

We've selected the free New Age Bootstrap theme. The theme is already packaged in a manner that will work with the theming tool.

Note: When being distributed, Plone themes are packaged as zip files. A theme should be structured such that there is only one top level directory in the root of the zip file. By convention the directory should contain your `index.html` and supporting files, the supporting files (CSS, javascript and other files) may be in subdirectories.

1. To get started download a copy of the New Age theme as a zip file. Then upload it to the theme controlpanel.

   Hint: This is a generic theme, it does not provide the Plone/Diazo specific `rules.xml` or `manifest.cfg` file. When you upload the zip file the theming tool generates a `rules.xml`. In the next steps you will add additional files including a `manifest.cfg` (perhaps in the future the manifest.cfg will also be generated for you).



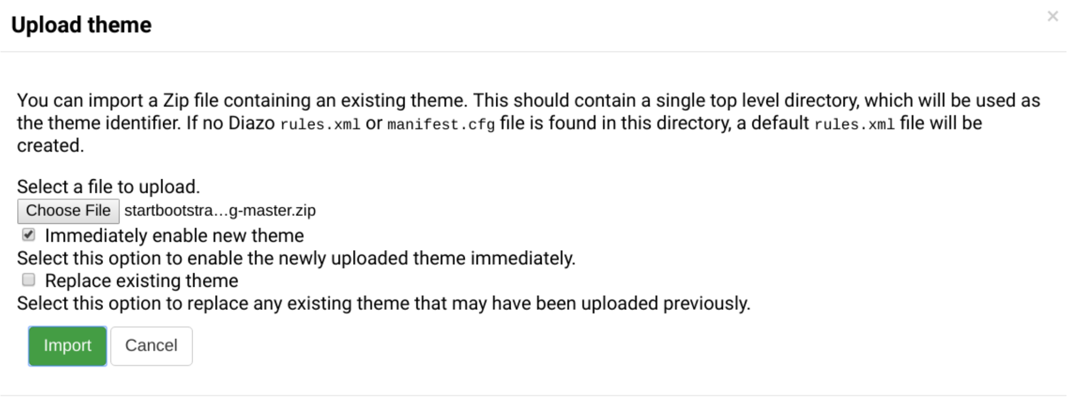
Themes

Use the buttons to create or upload a new Diazo theme, or sele

Barceloneta Theme

The default theme for Plone 5

Select the downloaded zip file.



Upload theme

You can import a Zip file containing an existing theme. This should contain a single top level directory, which will be used as the theme identifier. If no Diazo `rules.xml` or `manifest.cfg` file is found in this directory, a default `rules.xml` file will be created.

Select a file to upload.

Choose File startbootstra...g-master.zip

☑ Immediately enable new theme

Select this option to enable the newly uploaded theme immediately.

□ Replace existing theme

Select this option to replace any existing theme that may have been uploaded previously.

2. Add a `styles.less` file and import the Barceloneta styles.
3. Add a `manifest.cfg` file, set `production-css` equal to `styles.css`

Note: Clean Blog is a free Bootstrap theme, the latest version is available on github https://github.com/BlackrockDigital/startbootstrap-clean-blog

Hint: You can identify the theme path by reading your browser's address bar when your theme is open in the theming tool. You'll need to include the proper theme path in your `manifest.cfg`, in this case it will most likely be something like `++theme++startbootstrap-new-age-gh-pages`

[theme] title = New Age prefix = ++theme++startbootstrap-new-age-gh-pages/ production-css = ++theme++startbootstrap-new-age-gh-pages/styles.css

4. Add rules to include the Barceloneta backend utilities

```
  <?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" />
```

5. Add rules to include content, add site structure, drop unneeded elements, customize the menu.

Warning: Look out for inline styles in this theme (i.e. the use of the `style` attribute on a tag). This is especially problematic with background images set with relative paths. The two issues that result are:

- the relative path does not translate properly in the context of the theme;
- it can be tricky to dynamically replace background images provided by inline styles.

Creating a visitor-only theme - conditionally enabling Barceloneta

Sometimes it is more convenient for your website administrators to use Barceloneta, Plone's default theme. Other visitors would see a completely different layout provided by your custom theme. To achieve this you will need to associate your visitor theme rules with an expression like `css:if-content="body.userrole-anonymous"`. For rules that will affect logged-in users you can use the expression `css:if-content="body.:not(userrole-anonymous)"`.

Once you've combined the expressions above with the right Diazo rules you will be able to present an anonymous visitor with a specific HTML theme while presenting the Barceloneta theme to logged-in users.

Warning: The Barceloneta `++theme++barceloneta/rules.xml` expects the Barceloneta `index.html` to reside locally in your current theme. To avoid conflict and to accomodate the inherited Barceloneta, ensure that your theme file has a different name such as `front.html`.

Exercise 5 - Convert the theme to be a visitor-only theme

In this exercise we will alter our theme from the previous exercise to make it into a visitor-only theme.

1. Update the `rules.xml` file to include Barceloneta rules.

Hint: Use `<xi:include href="++theme++barceloneta/rules.xml" />`

2. Add conditional rules to `rules.xml` so that the new theme is only shown to anonymous users, rename the theme's `index.html` to `front.html` and add a copy of the Barceloneta `index.html`.

Hint: Copy the contents of the Barceloneta `index.html` file then add it to the theme as the new `index.html` file.

Change `rules.xml` to look similar to this:

```xml
<?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">

  <notheme css:if-not-content="#visual-portal-wrapper" />

  <rules css:if-content="body:not(.userrole-anonymous)">
    <!-- Import Barceloneta rules -->
    <xi:include href="++theme++barceloneta/rules.xml" />
  </rules>

  <rules css:if-content="body.userrole-anonymous">
    <theme href="front.html" />
    <replace css:theme-children=".intro header h2" css:content-
↪children=".documentFirstHeading" />
    <replace css:theme-children=".summary" css:content-children=".
↪documentDescription" />
    <replace css:theme-children=".preamble" css:content-children="
↪#content-core" />
  </rules>
</rules>
```

This page is included here from the *Mastering Plone Development*. The narrative is continued there.

Dexterity I: "Through The Web"

In this part you will:

- Create a new content type called *Talk*.

Topics covered:

- Content types
- Archetypes and Dexterity
- Fields
- Widgets

What is a content type?

A content type is a kind of object that can store information and is editable by users. We have different content types to reflect the different kinds of information about which we need to collect and display information. Pages, folders, events, news items, files (binary) and images are all content types.

It is common in developing a web site that you'll need customized versions of common content types, or perhaps even entirely new types.

Remember the requirements for our project? We wanted to be able to solicit and edit conference talks. We *could* use the **Page** content type for that purpose. But we need to make sure we collect certain bits of information about a talk and we couldn't be sure to get that information if we just asked potential presenters to create a page. Also, we'll want to be able to display talks featuring that special information, and we'll want to be able to show collections of talks. A custom content type will be ideal.

The makings of a Plone content type

Every Plone content type has the following parts:

Schema A definition of fields that comprise a content type; properties of an object.

FTI The "Factory Type Information" configures the content type in Plone, assigns it a name, an icon, additional features and possible views to it.

Views A view is a representation of the object and the content of its fields that may be rendered in response to a request. You may have *one or more* views for an object. Some may be *visual* — intended for display as web pages — others may be intended to satisfy AJAX requests and render content in formats like JSON or XML.

Dexterity and Archetypes - A Comparison

There are two content frameworks in Plone:

- *Dexterity*: new and the coming default.
- *Archetypes*: old, tried and tested. Widespread, used in many add-ons.
- Plone 4.x: Archetypes is the default, with Dexterity available.
- Plone 5.x: Dexterity is the default, with Archetypes available.
- For both, add and edit forms are created automatically from a schema.

What are the differences?

- Dexterity: New, faster, modular, no dark magic for getters and setters.
- Archetypes had magic setter/getter - use `talk.getAudience()` for the field `audience`.
- Dexterity: fields are attributes: `talk.audience` instead of `talk.getAudience()`.

"Through The Web" or TTW, i.e. in the browser, without programming:

- Dexterity has a good TTW story.
- Archetypes has no TTW story.
- UML-modeling: ArchGenXML for Archetypes, agx for Dexterity

Approaches for Developers:

- Schema in Dexterity: TTW, XML, Python. Interface = schema, often no class needed.
- Schema in Archetypes: Schema only in Python.

- Dexterity: Easy permissions per field, easy custom forms.
- Archetypes: Permissions per field are hard, custom forms even harder.
- If you have to program for old sites you need to know Archetypes!
- If starting fresh, go with Dexterity.

Extending:

- Dexterity has Behaviors: easily extendable. Just activate a behavior TTW and your content type is e.g. translatable (`plone.app.multilingual`).
- Archetypes has `archetypes.schemaextender`. Powerful but not as flexible.

We have only used Dexterity for the last few years. We teach Dexterity and not Archetypes because it's more accessible to beginners, has a great TTW story and is the future.

Views:

- Both Dexterity and Archetypes have a default view for content types.
- Browser Views provide custom views.
- You can generate views for content types in the browser without programming (using the `plone.app.mosaic` Add-on).
- Display Forms.

Modifying existing types

- Go to the control panel http://localhost:8080/Plone/@@dexterity-types
- Inspect some of the existing default types.
- Select the type *News Item* and add a new field `Hot News` of type *Yes/No*
- In another tab, add a *News Item* and you'll see the new field.
- Go back to the schema-editor and click on Edit XML Field Model.
- Note that the only field in the XML schema of the News Item is the one we just added. All others are provided by behaviors.
- Edit the form-widget-type so it says:

```
<form:widget type="z3c.form.browser.checkbox.SingleCheckBoxFieldWidget"/>
```

- Edit the News Item again. The widget changed from a radio field to a check box.
- The new field `Hot News` is not displayed when rendering the News Item. We'll take care of this later.

See also:

http://docs.plone.org/external/plone.app.contenttypes/docs/README.html#extending-the-types

Creating content types TTW

In this step we will create a content type called *Talk* and try it out. When it's ready we will move the code from the web to the file system and into our own add-on. Later we will extend that type, add behaviors and a viewlet for Talks.

- Add new content type "Talk" and some fields for it:
  - *Add new field* "Type of talk", type "Choice". Add options: talk, keynote, training.

  - *Add new field* “Details”, type “Rich Text” with a maximal length of 2000.
  - *Add new field* “Audience”, type “Multiple Choice”. Add options: beginner, advanced, pro.
  - Check the behaviors that are enabled: *Dublin Core metadata*, *Name from title*. Do we need them all?
- Test the content type.
- Return to the control panel http://localhost:8080/Plone/@@dexterity-types
- Extend the new type: add the following fields:
  - “Speaker”, type: “Text line”
  - “Email”, type: “Email”
  - “Image”, type: “Image”, not required
  - “Speaker Biography”, type: “Rich Text”
- Test again.

Here is the complete XML schema created by our actions:

```
<model xmlns:lingua="http://namespaces.plone.org/supermodel/lingua"
     xmlns:users="http://namespaces.plone.org/supermodel/users"
     xmlns:security="http://namespaces.plone.org/supermodel/security"
     xmlns:marshal="http://namespaces.plone.org/supermodel/marshal"
     xmlns:form="http://namespaces.plone.org/supermodel/form"
     xmlns="http://namespaces.plone.org/supermodel/schema">
  <schema>
    <field name="type_of_talk" type="zope.schema.Choice">
      <description/>
      <title>Type of talk</title>
      <values>
        <element>Talk</element>
        <element>Training</element>
        <element>Keynote</element>
      </values>
    </field>
    <field name="details" type="plone.app.textfield.RichText">
      <description>Add a short description of the talk (max. 2000 characters)</description>
      <max_length>2000</max_length>
      <title>Details</title>
    </field>
    <field name="audience" type="zope.schema.Set">
      <description/>
      <title>Audience</title>
      <value_type type="zope.schema.Choice">
        <values>
          <element>Beginner</element>
          <element>Advanced</element>
          <element>Professionals</element>
        </values>
      </value_type>
    </field>
    <field name="speaker" type="zope.schema.TextLine">
      <description>Name (or names) of the speaker</description>
      <title>Speaker</title>
    </field>
    <field name="email" type="plone.schema.email.Email">
      <description>Adress of the speaker</description>
      <title>Email</title>
    </field>
    <field name="image" type="plone.namedfile.field.NamedBlobImage">
      <description/>
      <required>False</required>
      <title>Image</title>
    </field>
    <field name="speaker_biography" type="plone.app.textfield.RichText">
      <description/>
      <max_length>1000</max_length>
      <required>False</required>
      <title>Speaker Biography</title>
    </field>
  </schema>
</model>
```

Moving contenttypes into code

It’s awesome that we can do so much through the web. But it’s also a dead end if we want to reuse this content type in other sites.

Also, for professional development, we want to be able to use version control for our work, and we’ll want to be able to add the kind of business logic that will require programming.

So, we’ll ultimately want to move our new content type into a Python package. We’re missing some skills to do that, and we’ll cover those in the next couple of chapters.

See also:

- Dexterity Developer Manual
- The standard behaviors

Exercises

Exercise 1

Modify Pages to allow uploading an image as decoration (like News Items do).

Solution

- Go to the dexterity control panel (http://localhost:8080/Plone/@@dexterity-types)
- Click on *Page* (http://127.0.0.1:8080/Plone/dexterity-types/Document)
- Select the tab *Behaviors* (http://127.0.0.1:8080/Plone/dexterity-types/Document/@@behaviors)
- Check the box next to *Lead Image* and save.

The images are displayed above the title.

Exercise 2

Create a new content type called *Speaker* and export the schema to a XML File. It should contain the following fields:

- Title, type: "Text Line"
- Email, type: "Email"
- Homepage, type: "URL" (optional)
- Biography, type: "Rich Text" (optional)
- Company, type: "Text Line" (optional)
- Twitter Handle, type: "Text Line" (optional)
- IRC Handle, type: "Text Line" (optional)
- Image, type: "Image" (optional)

Do not use the DublinCore or the Basic behavior since a speaker should not have a description (unselect it in the Behaviors tab).

We could use this content type later to convert speakers into Plone users. We could then link them to their talks.

Solution

The schema should look like this:

```
<model xmlns:lingua="http://namespaces.plone.org/supermodel/lingua"
       xmlns:users="http://namespaces.plone.org/supermodel/users"
       xmlns:security="http://namespaces.plone.org/supermodel/security"
       xmlns:marshal="http://namespaces.plone.org/supermodel/marshal"
       xmlns:form="http://namespaces.plone.org/supermodel/form"
       xmlns="http://namespaces.plone.org/supermodel/schema">
  <schema>
    <field name="title" type="zope.schema.TextLine">
      <title>Name</title>
    </field>
    <field name="email" type="plone.schema.email.Email">
      <title>Email</title>
    </field>
    <field name="homepage" type="zope.schema.URI">
      <required>False</required>
      <title>Homepage</title>
    </field>
    <field name="biography" type="plone.app.textfield.RichText">
      <required>False</required>
      <title>Biography</title>
    </field>
    <field name="company" type="zope.schema.TextLine">
      <required>False</required>
      <title>Company</title>
    </field>
    <field name="twitter_handle" type="zope.schema.TextLine">
      <required>False</required>
      <title>Twitter Handle</title>
    </field>
    <field name="irc_name" type="zope.schema.TextLine">
      <required>False</required>
      <title>IRC Handle</title>
    </field>
    <field name="image" type="plone.namedfile.field.NamedBlobImage">
      <required>False</required>
      <title>Image</title>
    </field>
  </schema>
</model>
```

See also:

- Dexterity XML
- Model-driven types

In one of the next chapters of *Mastering Plone Development* it is explained how you can move the content-type you created into code: *Return to Dexterity: Moving contenttypes into Code*.

Mosaic

In this section we will:

- create a *home page* layout,
- create a *specific talk detail* layout.

Topics covered:

- Create custom layouts.
- Manage layouts.
- Use the layout editor.

What is Mosaic?

- A Plone add-on,
- which allows managing layouts from the Plone interface.

Some comparisons

- Compared to Diazo:

  Diazo enables theming our Plone site by providing CSS, images, and HTML templates. It will apply to the entire page (footer, main content, portlets, etc.).

  Mosaic uses the grid provided by our design to dynamically build specific content layouts.

- Compared to `collective.cover`:

  `collective.cover` provides a specific content-type (a "Cover page") where we can manage the layout in order to build our homepage.

  Mosaic does not provide any content-type, it allows to edit any existing content layout.

Installation

On an existing Plone Buildout

If you already have your own Plone installation you can install Mosaic by customizing it as follows:

Modify `buildout.cfg` to add Mosaic as a dependency:

```
eggs =
    ...
    plone.app.mosaic

versions =
    ...
    plone.tiles = 1.8.0
    plone.subrequest = 1.7.0
    plone.app.tiles = 3.0.0
    plone.app.standardtiles = 2.0.0rc1
    plone.app.blocks = 4.0.0rc1
    plone.app.drafts = 1.1.1
    plone.app.mosaic = 2.0.0rc1
```

Run your buildout:

```
$ bin/buildout -N
```

Then go to *Site Setup* → *Add-ons* and Mosaic *Install*.

We will use a Plone pre-configured Heroku instance.

Once deployed, create a Plone site, then go to the *Site Setup* → *Add-ons* and Mosaic *Install*.

Principle

The basic component of a Mosaic based layout is called a tile. A layout is a combination of several tiles.

A tile is a dynamic portion of a web page, it can be a text element, an image, a field, etc.

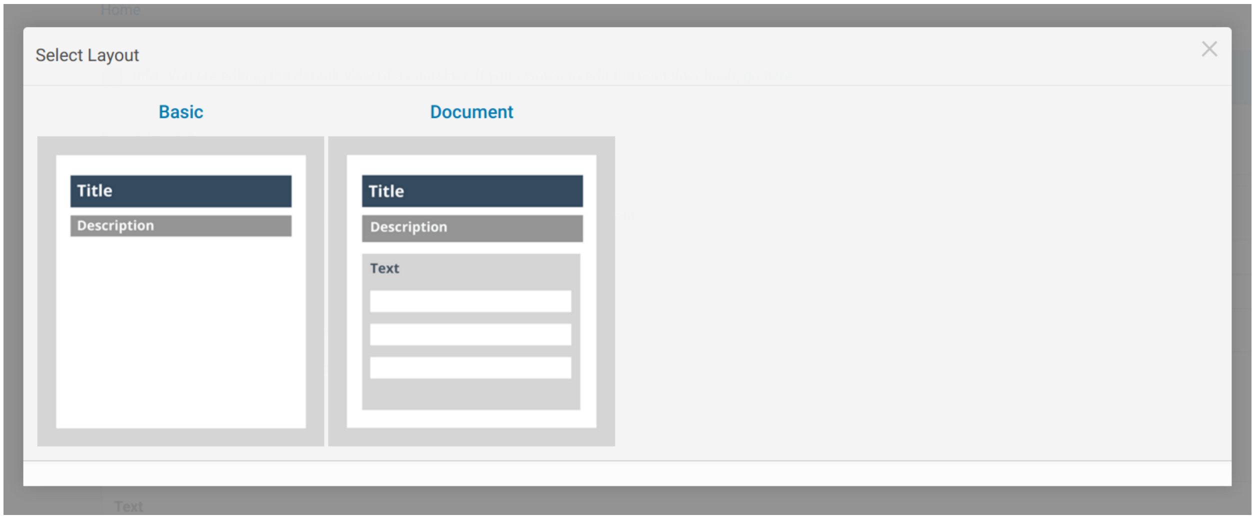
Mosaic provides an editor able to easily position tiles across our theme's grid.

The Mosaic editor

To enable the Mosaic editor on a content item change its default display as follows: go to *Display* → *Mosaic layout*.

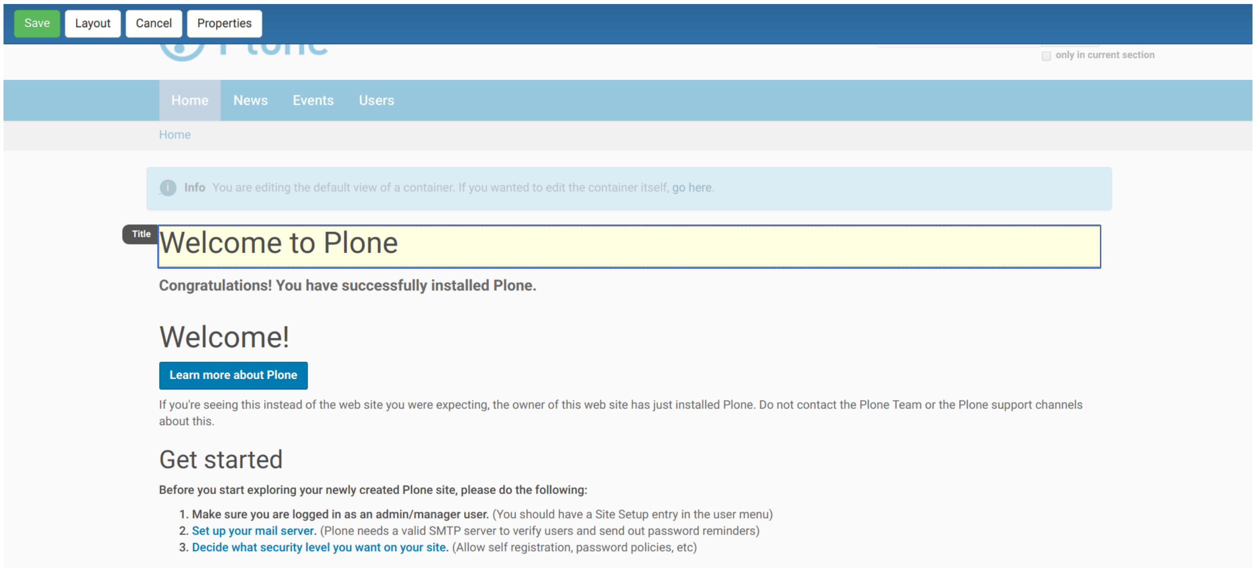
You have now enabled the Mosaic editor as a replacement for the default edit form.

Click on *Edit* if this is the first time editing the current item you will be prompted to select a layout.



Choose a layout.

This editor allows to change our content fields content (just like the regular Plone form), but the fields are rendered into the view layout and they are edited in-place.



Info You are editing the default view of a container. If you wanted to edit the container itself, go here.

Welcome to Plone

Congratulations! You have successfully installed Plone.

Welcome!

If you're seeing this instead of the web site you were expecting, the owner of this web site has just installed Plone. Do not contact the Plone Team or the Plone support channels about this.

Get started

Before you start exploring your newly created Plone site, please do the following:

1. Make sure you are logged in as an admin/manager user. (You should have a Site Setup entry in the user menu)
2. Set up your mail server. (Plone needs a valid SMTP server to verify users and send out password reminders)
3. Decide what security level you want on your site. (Allow self registration, password policies, etc)

The top bar offers different buttons:

- *Save*, to save our field entries.
- *Cancel*, to cancel our changes.
- *Properties*, to access the content properties: it displays the regular Plone form tabs, but the fields currently involved in the layout are hidden.
- *Layout*, to manage the content layout.

Exercise 1 - Change the layout of the front page

Go to the front page of the website and update the layout as follows:

1. Activate *Display* → *Mosaic layout*

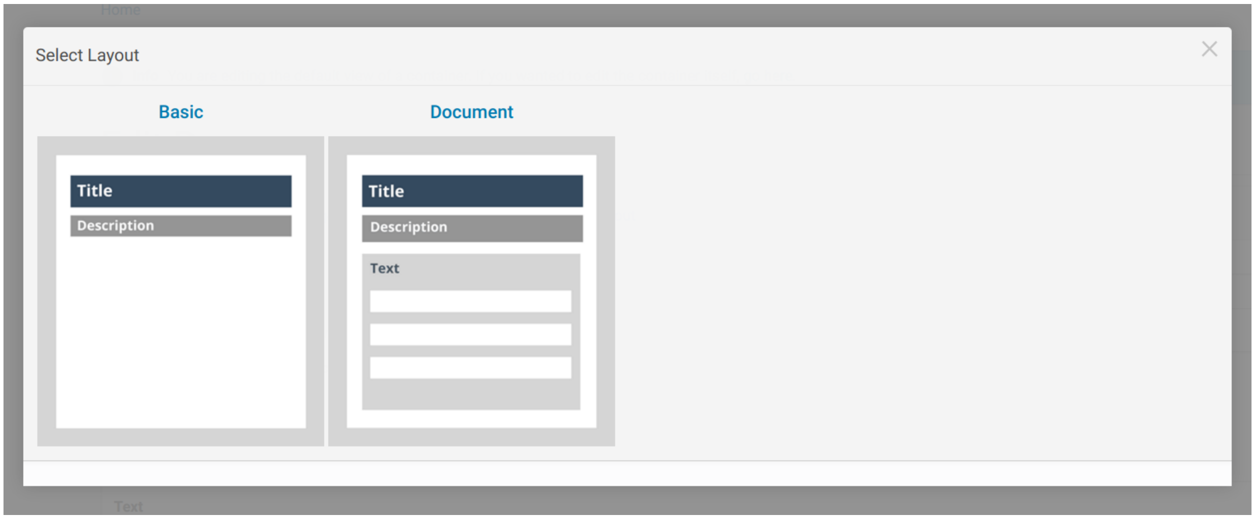
2. *Edit* and select the "Document" layout
3. The select *Layout* → *Customize*
4. Add a Document Byline to the bottom of the layout *Insert* > *Document Byline*
5. Click *Save*

In the context of the Mosaic Editor, do you know the difference between *Save* and *Layout* → *Save*?

Change the content layout

If we click on *Layout* → *Change*, we can choose the layout we want for our content. The choices are restricted to the layout applicable to the current content-type.

For instance for a Page, Mosaic proposes (by default) two layouts: Basic and Document.



Customize a content layout

If we click on *Layout* → *Customize*, the Mosaic editor switches to the layout mode, where we can still change our field values, but also change the layout:

- by hovering the page content, existing tiles are highlighted and we can drag & drop them in different places,
- by clicking in a tile, we can edit its content,
- by clicking outside the curently edited tile, we disable the edit mode.

In layout mode, the top bar contains two extra buttons:

- *Format*, which provides different simple formatting options for tiles (text padding, floating) or for rows (change background color),
- *Insert*, which allows to add new tiles to our layout.

The tiles

Mosaic provides the following tiles:

- Structure tiles:
  - heading,

  - subheading,
  - text,
  - table,
  - bulleted list,
  - numbered list,
  - table of contents,
  - navigation: this tiles displays a navigation menu, its settings can be changed in a modal window (click on the “i” button on the bottom-right corner to display the modal),
- Media:
  - image,
  - embed: it allows to display any remote embeddable content (like a YouTube video for instance),
  - attachment,
- Fields: all the existing fields of the current content,
- Applications: for now, there is only Discussion, which shows the discussion form (discussion needs to be enable in the site setup),
- Properties:
  - document byline,
  - related contents,
  - keywords,
- Advanced:
  - content listing: it is a collection-like tile, it allows to list all contents matching given criterias (criterias can be changed in the modal window),
  - existing content: it allows to display another content in a tile
  - if Rapido is installed, there is also a Rapido tile, which allows to display any Rapido block.

Exercise 2: Customize the home page layout

Create an attractive layout for the home page.

Solution

- go to Display menu and select “Mosaic layout”,
- click Edit,
- click on *Layout* → *Customize*,
- change the layout,
- click Save.

Create a reusable layout

When the layout has been customized, the *Layout* menu offers a *Save* action.

This action allows to save the current layout as a reusable layout.

If `Global` is checked, the layout will be usable by any user (else it is restricted to the current user).

The layout is associated to the current content type, by default it will not be usable for other content types.

Once saved, our layout will be listed with the other available layouts when we click on *Layout* → *Change*.

Exercise 3: create a layout for talks

Note: This exercise assumes that you have created a content type called "Talk". you can quickly create one by the following the steps in Dexterity: Creating TTW content types documentation.

Create an attractive layout for a talk, save it and reuse it for another talk.

Solution

- customize a talk layout (see Exercise 2),
- click on :menuselection:*Layout* → *Save*,
- enter its title: "Talk", and select "Global",
- click *Save*,
- navigate to another talk,
- go to *Display* menu and select "Mosaic layout",
- click *Edit*,
- click on Layout / Change,
- choose "Talk".

Manage custom layouts

Custom layouts can be managed from the Plone control panel:

- click on *user menu* → *Site settings*,
- click on Mosaic Layout Editor (in the last section, named *Add-on configuration*),

In the third tab of this control panel, named "Show/hide content layouts", we can see the existing layouts, their associated content types, and we can deactivate (or re-activate) them by clicking on *Hide* (or *Show*).

In the first tab, named *Content layouts*, there is a source editor.

By editing `manifest.cfg`, we can assign a layout to another content type by changing the `for =` line. If we remove this line, the layout is available for any content type.

We can also delete the layout section from `manifest.cfg`, and the layout will be deleted (if we do so, it is recommended to delete its associated HTML file too).

Deleting a custom layout can also be managed in another way:

Note: the second tab, named *Site layouts*, is not usable for now.

Edit the layout HTML structure

In the Mosaic Layout Editor's first tab ("Content layouts"), `manifest.cfg` is not the only editable file.

There is also some HTML files. Each of them corresponds to a layout and they represent what we have built by drag&dropping tiles in our layouts.

Using the code editor, we can change this HTML structure manually instead of using the WYSIWIG editor.

Layouts are implemented in regular HTML using nested `<div>` elements and specific CSS classes. Those classes are provided by the Mosaic grid which works like any CSS grid:

- **structure:**
  - `mosaic-grid-row`
  - `mosaic-grid-cell`
- **sizes:**
  - `mosaic-width-full`
  - `mosaic-width-half`
  - `mosaic-width-quarter`
  - `mosaic-width-three-quarters`
  - `mosaic-width-third`
  - `mosaic-width-two-thirds`
- **positions:**
  - `mosaic-position-leftmost`
  - `mosaic-position-third`
  - `mosaic-position-two-thirds`
  - `mosaic-position-quarter`
  - `mosaic-position-half`
  - `mosaic-position-three-quarters`

Import layouts

We might want to work on a layout on our development server, and then be able to deploy it on our production server.

We can achieve that using the Mosaic editor control panel, which allows to copy the layout HTML structure and its declaration in `manifest.cfg`.

Rapido

In this part you will:

- Create a *Like* button on any talk so that visitors can cast votes,
- Display the total of votes next to the button,
- Create a "Top 5" page,

- Reset the votes on workflow change.

Topics covered:

- Create a Rapido app.
- Insert Rapido blocks in Plone pages.
- Implement scripts in Rapido.

What is Rapido?

Rapido is a Plone add-on that allows implementation of custom features on top of Plone. It is a simple yet powerful way to extend the behavior of your Plone site without using the underlying frameworks. The **Plone theming tool** is the interface used to build `rapido.plone` applications. This means that Rapido applications can be written both **on the file system** or using the **inline editor** of the Plone theming tool.

A Rapido application is just a part of your current theme: It can be imported, exported, copied, modified, etc. just like the rest of the theme. But in addition to layout and design elements, it can contain business logic implemented in Python.

A couple of comparisons

- Compared to **Dexterity**:
  - Dexterity focuses on content types. Content types can only use the Plone business logic, you cannot implement your own logic.
  - By contrast, using Rapido you can implement your own logic; however you can only store data records, not **Plone content items** (at least, not directly like Dexterity does).
- Compared to **Diazo** and **Mosaic**:
  - Diazo manages the Plone theme,
  - Mosaic allows you to manage layouts by positioning tiles,
  - Rapido does not do either theming or layouts, but a Rapido block can be called from a Diazo rule or displayed in a Mosaic tile.
- Compared to conventional Plone development:
  - Rapido is simpler: no need to learn about any framework, no need to create Python eggs,
  - but Rapido code runs in restricted mode, so you cannot import any unsafe Python module in your code.

Installation

For the training, we will use a Heroku instance pre-configured with Plone.

Once deployed:

- create a Plone site,
- go to: *Plone control panel* -> *Add-ons* (http://localhost:8080/Plone/prefs_install_products_form),
- finally: install Rapido.

But to deploy Rapido on an actual Plone instance, modify `buildout.cfg` to add Rapido as a dependency:

```
eggs =
    ...
    rapido.plone
```

Run your buildout:

```
$ bin/buildout -N
```

Then go to *Plone control panel* -> *Add-ons* `http://localhost:8080/Plone/prefs_install_products_form`, and install Rapido.

Principles

Rapido *application* It contains the features you implement; it is just a folder containing templates, Python code, and YAML files.

block Blocks display a chunk of HTML which can be inserted in your Plone pages.

element Elements are the dynamic components of your blocks. They can be input fields, buttons, or just computed HTML. They can also return JSON if you call them from a javascript app,

records A Rapido app is able to store data as records. Records are just basic dictionaries.

How to create a Rapido app

A Rapido app is defined by a set of files in our Diazo theme.

The files need to be in a specific location:

```
/rapido/<app-name>
```

Here is a typical layout for a Rapido app:

```
/rapido
    /myapp
        settings.yaml
        /blocks
            stats.html
            stats.py
            stats.yaml
            tags.html
            tags.py
            tags.yaml
```

Todo

ADD SCREENSHOT HERE

Blocks and elements

The app components are *blocks*. A block is defined by a set of 3 files (HTML, Python, and YAML) located in the `blocks` folder.

The **YAML file** defines the *elements*. An *element* is any dynamically generated element in a block. It can be a form field (input, select, etc.), or a button (an `ACTION` element), or even just a piece of generated HTML (a `BASIC` element).

The **HTML file** contains the layout of the block. The templating mechanism is super simple: elements are simply enclosed in curly brackets, like this: `{my_element}`.

The **Python file** contains the application logic. We will see later how exactly we use those Python files.

Exercise 1: Create the vote block

Let's start by displaying a static counter showing "0 votes" on all talks.

First, we need to create the `rating` Rapido app.

Solution

- Go to the Plone theming control panel: http://localhost:8080/Plone/@@theming-controlpanel
- Copy the Barceloneta theme, name it `training` and enable it immediately,
- Add a new folder named `rapido`,
- And add a subfolder named `rating`.

The Rapido app is initialized.

And now, we need to create a `rate` block.

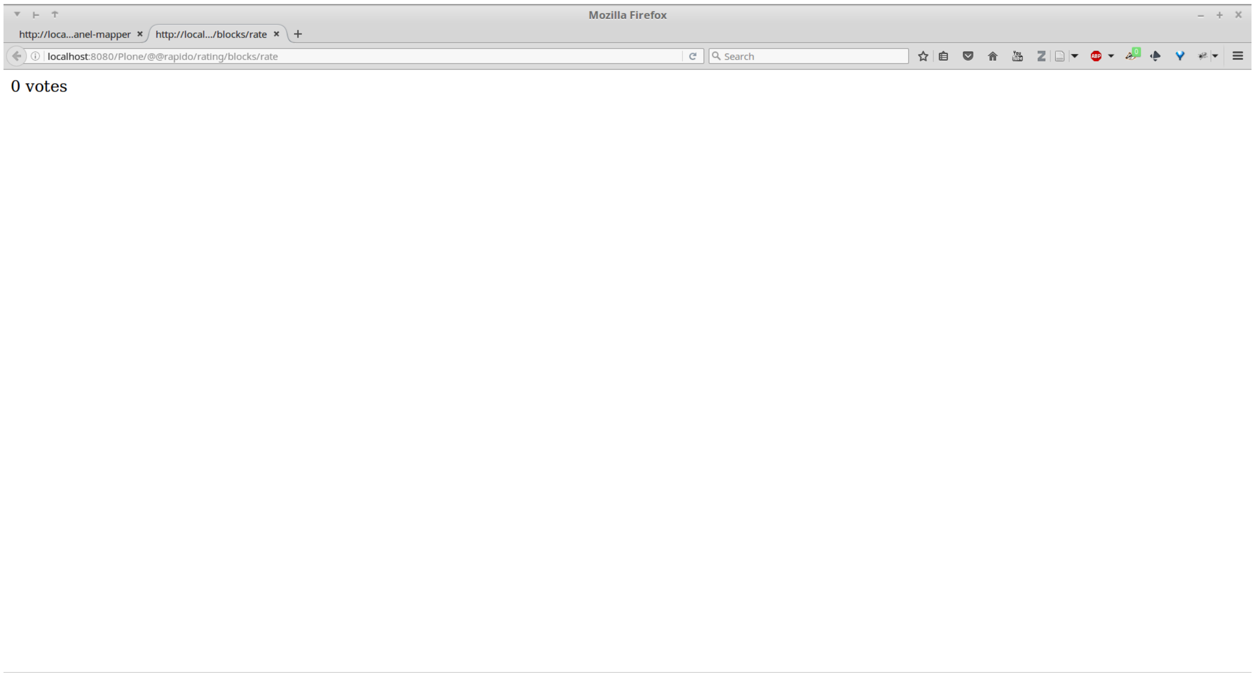
Solution

- Add a folder named `blocks` in `rating`,
- In `blocks`, add a file named `rate.html`,
- In the file, put the following content:

```
<span>0 votes</span>
```

Once the block is ready, you can display it by visiting its URL in your browser:

http://localhost:8080/Plone/@@rapido/rating/blocks/rate


Mozilla Firefox
http://loca...anel-mapper
http://local.../blocks/rate
localhost:8080/Plone/@@rapido/rating/blocks/rate
Search

0 votes

But we would prefer to display it inside our existing Plone pages.

Include Rapido blocks in Plone pages

We can include Rapido blocks in Plone pages using Diazo rules.

The `include` rule is able to load another URL than the current page, extract a piece of HTML from it, and include it in regular Diazo rules (such as `after`, `before`, etc.).

So the following rule:

```
<after css:content="#content">
    <include href="@@rapido/stats/blocks/stats" css:content="form"/>
</after>
```

would insert the `stats` block under the Plone main content.

Rapido rules can be added directly in our theme's main `rules.xml` file, but it is a good practice to put them in a dedicated rule file which can be located in our app folder.

The app-specific rules file can be included in the main rules file as follows:

```
<xi:include href="rapido/myapp/rules.xml" />
```

Exercise 2: Display the vote block in Plone pages

Insert the `rate` block content under the Plone page main heading.

Solution

- in the main `rules.xml`, add the following line just after the first `<rules>` opening tag:

```
<xi:include href="rapido/rating/rules.xml" />
```

- In the `rating` folder, add a new file named `rules.xml` containing:

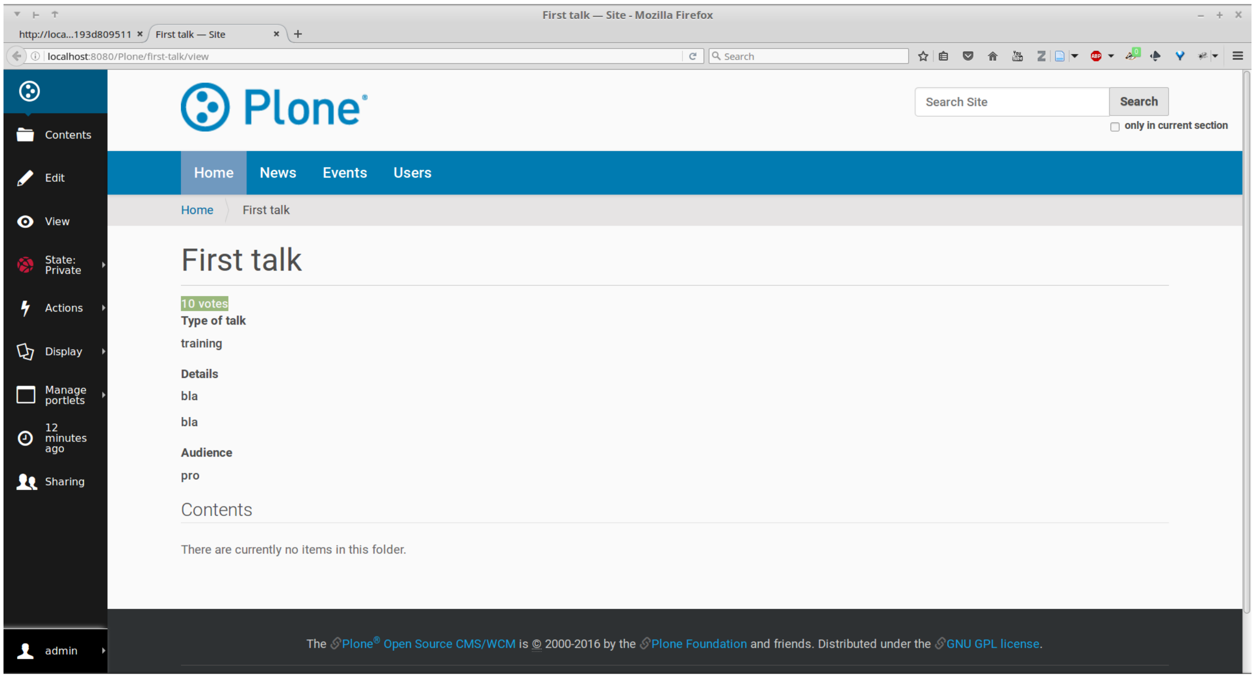
```
<?xml version="1.0" encoding="utf-8"?>
<rules xmlns="http://namespaces.plone.org/diazo"
       xmlns:css="http://namespaces.plone.org/diazo/css"
       xmlns:xhtml="http://www.w3.org/1999/xhtml"
       xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
       xmlns:xi="http://www.w3.org/2001/XInclude">

    <after css:content=".documentFirstHeading" css:if-content=".template-view.
↪portaltype-talk">
        <include href="@@rapido/rating/blocks/rate" css:content="form"/>
    </after>

</rules>
```

Let's detail what it does:

- the `after` rule targets the page heading (identified by the `.documentFirstHeading` selector), but it only applies when we are viewing a talk (`.template-view.portaltype-talk`),
- the `include` rule retrieves the Rapido block content.

Note: This presumes having completed *Dexterity I: "Through The Web"*.

Now, if you visit a talk page, you see the counter below the heading.



First talk — Site - Mozilla Firefox

localhost:8080/Plone/first-talk/view

First talk

10 votes

Type of talk

training

Details

bla

bla

Audience

pro

Contents

There are currently no items in this folder.

The Plone® Open Source CMS/WCM is © 2000-2016 by the Plone Foundation and friends. Distributed under the GNU GPL license.

Make our blocks dynamic

The YAML file allows us to declare elements. The Python files allows computing the *element value* using a function named after the element id. And the HTML file can display elements using the curly-brackets notation. The 3 files must have the same name (only the extensions change).

As mentioned earlier, the **Python file** contains the application logic.

This file is a set of Python functions named to correspond to the elements or the events they relate to.

For a `BASIC` element for instance, if we provide a function with the same name as the element, its return-value will be inserted in the block at the location of the element.

For an `ACTION`, if we provide a function with the same name as the element, it will be executed when a user clicks on the action button.

A typical element is defined and used as follows:

- create a definition in the YAML file:

```
elements:
    answer:
        type: BASIC
```

- create an implementation in the Python file:

```
def answer(context):
    return 42
```

- insert the element in the HTML template:

```
<span>Answer to the Ultimate Question of Life, the Universe, and Everything:
↪{answer}</span>
```

Exercise 3: Create an element to display the votes

Let's replace the “0” value in our rate block with a computed value.

To do this, you need to add an element to the block. For now the Python function will just return 10.

Solution

- In the `blocks` folder, add a new file named `rate.yaml` containing:

```
elements:
    display_votes:
        type: BASIC
```

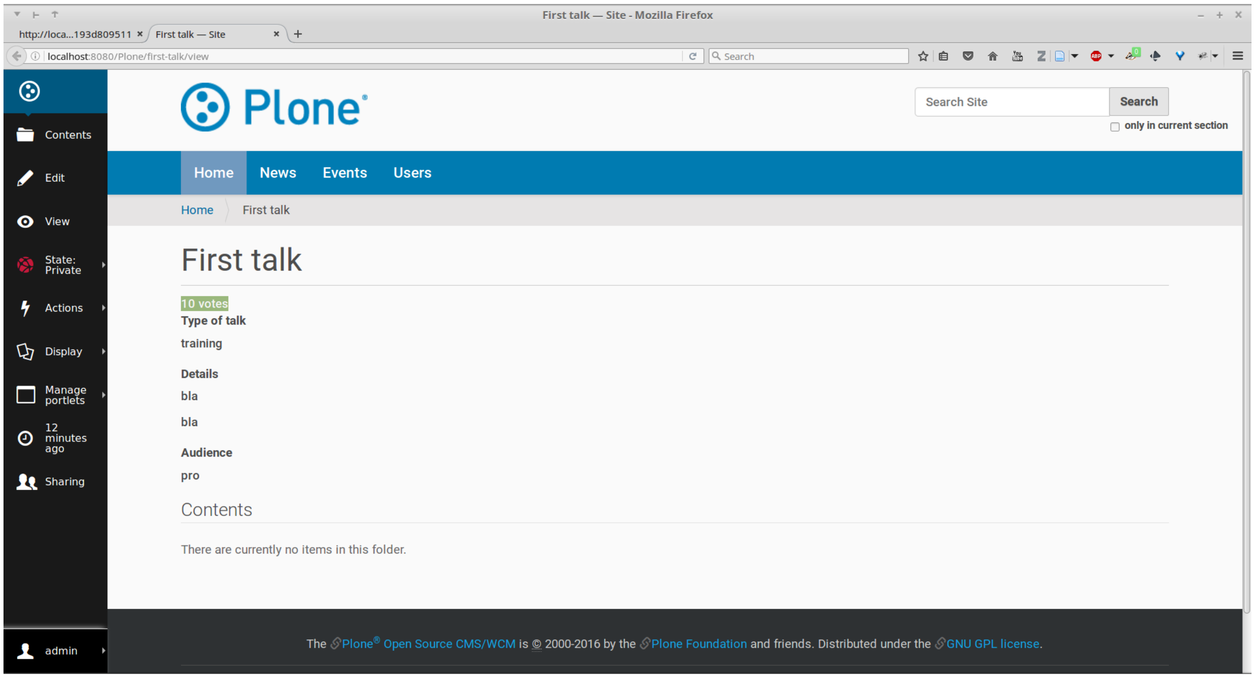
- Add also a file named `rate.py` containing:

```
def display_votes(context):
    return 10
```

- And change the existing `rate.html` as follows:

```
<span>{display_votes} votes</span>
```

Now, if you refresh your talk page, the counter will display the value returned by your Python function.



First talk

10 votes

Type of talk

training

Details

bla

bla

Audience

pro

Contents

There are currently no items in this folder.

The Plone® Open Source CMS/WCM is © 2000-2016 by the Plone Foundation and friends. Distributed under the GNU GPL license.

Create actions

An *action* is a regular element, but it is rendered as a button.

Its associated function in the Python file will be called when the user clicks on the button.

Example:

- YAML:

```
elements:
    change_page_title:
        type: ACTION
        label: Change the title
```

- Python:

```
def change_page_title(context):
    context.content.title = "A new title"
```

- HTML:

```
<span>{change_page_title}</span>
```

Every time the user clicks the action, the block is reloaded (so elements are refreshed).

When the block is inserted in a Plone page using a Diazo rule, the reloading will just replace the current page with the bare block. Usually this is not what we want. If we want the current Plone page to be preserved, we need to activate the AJAX mode in the YAML file:

```
target: ajax
```

Exercise 4: Add the Like button

Add a *Like* button to the block. For now, the action itself will do nothing. Let's just insert it at the right place, and make sure the block is refreshed properly when we click.

Solution

- in `rate.yaml`, add a new `like` element and change the target to `ajax` After doing this, your YAML file looks as follows:

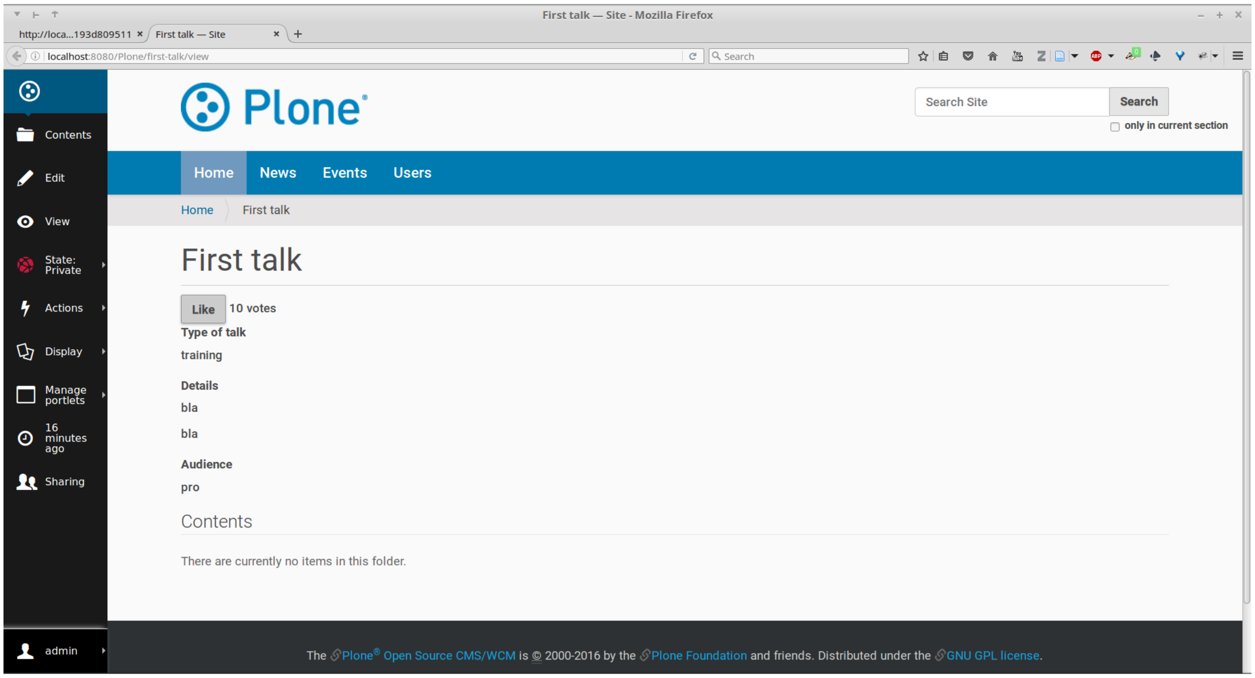
```
target: ajax
elements:
    display_votes:
        type: BASIC
    like:
        type: ACTION
        label: Like
```

- in `rate.py`, add a new function:

```
def like(context):
    # do nothing for now
    pass
```

- and in `rate.html`:

```
<span>{like} {display_votes} votes</span>
```



First talk

10 votes

Type of talk

training

Details

bla

bla

Audience

pro

Contents

There are currently no items in this folder.

The Plone® Open Source CMS/WCM is © 2000-2016 by the Plone Foundation and friends. Distributed under the GNU GPL license.

Store data

Each Rapido app provides an internal storage utility able to store records.

Records are not Plone objects, they are just simple dictionaries of basic data (strings, numbers, dates, etc.). There is no constraint on the dictionary items but Rapido will always set an `id` item, so this key is reserved.

Something like:

```
{'id': 'record_1', 'name': 'Eric', 'age': 42}
```

could be a valid record.

The Rapido Python API allows us to create, get or delete records:

```
record = context.app.create_record(id="my-record")
record = context.app.get_record("other-record")
context.app.delete_record("other-record")
```

The record items are managed like regular Python dictionary items:

```
record.get('age', 0)
'age' in record
record['age'] = 42
del record['age']
```

Exercise 5: Count votes

The button is OK now, now let's focus on counting votes. To count the votes on a talk, you need store some information:

- an identifier for the talk (we will use the talk path, from the Plone `absolute_url_path()` method),
- the total votes it gets.

Let's implement the `like()` function:

- first we need to get the current talk: the Rapido `context` allows to get the current Plone content using `context.content`,
- then we need to get the record corresponding to the current talk, - if it does not exist, we need to create it,
- and then we need to increase the current total votes for that talk by 1.

Solution

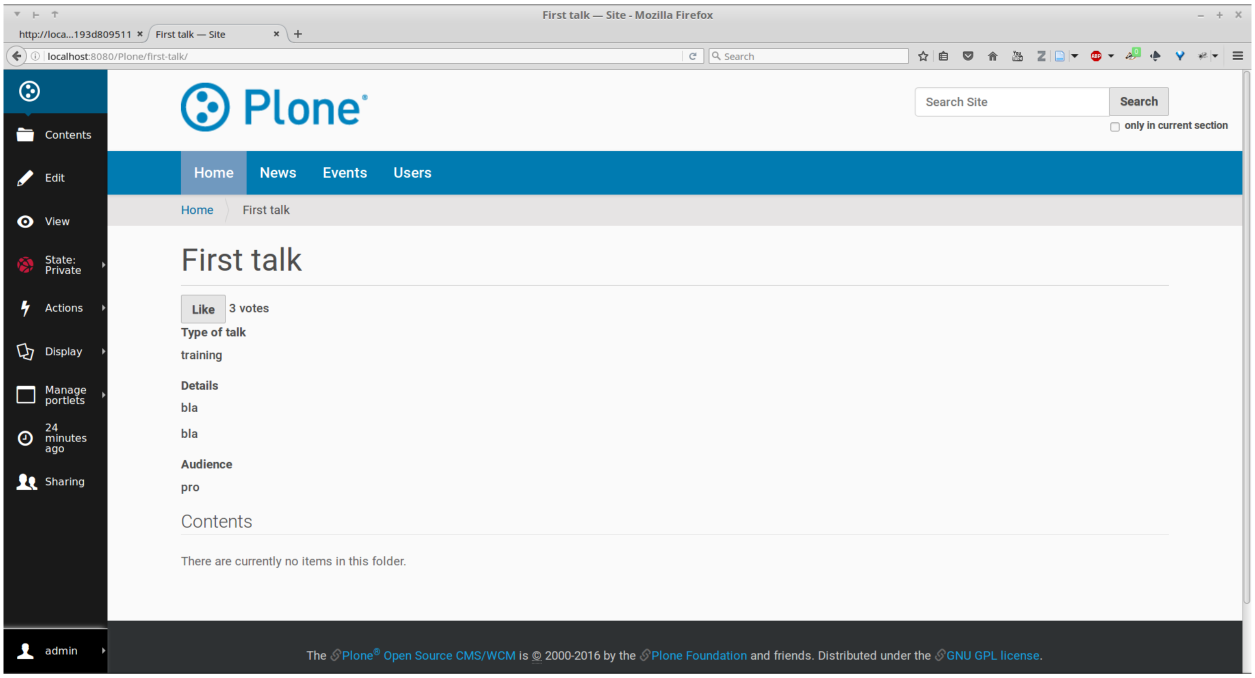
```
def like(context):
    current_talk = context.content
    talk_path = current_talk.absolute_url_path()
    record = context.app.get_record(talk_path)
    if not record:
        record = context.app.create_record(id=talk_path)
        record['total'] = 0
    record['total'] += 1
```

Note: we cannot just use the content `id` attribute as a valid identifier because it is not unique at site level, so we prefer the path.

Now let's make sure to display the proper total in the `display_votes` element:

- here also, we need to get the current talk,
- then we get the corresponding record,
- and we get its current total votes.

```
def display_votes(context):
    talk_path = context.content.absolute_url_path()
    record = context.app.get_record(talk_path)
    if not record:
        return 0
    return record['total']
```



First talk

3 votes

Type of talk

training

Details

bla

bla

Audience

pro

Contents

There are currently no items in this folder.

The Plone® Open Source CMS/WCM is © 2000-2016 by the Plone Foundation and friends. Distributed under the GNU GPL license.

HTML templating vs TAL templating

HTML templating

The Rapido HTML templating is very simple. It is just plain HTML with curly-bracket notations to insert elements:

```
<p>This is my message: {message}</p>
```

If the element is an object, we can render its properties:

```
def doc(context):
    return context.content
```

```
<p>This is my title: {doc.title}</p>
```

And if the element is a dictionary, we can access its items:

```
def stats(context):
    return {'avg': 10, 'total': 120}
```

```
<p>Average: {stats[avg]}</p>
```

It is easy to use but it cannot perform loops or conditional insertion.

TAL templating

TAL templating is the templating format used in the core of Plone. If HTML templating is too limiting, Rapido allows you to use TAL instead.

We just need to provide a file with the `.pt` extension instead of the HTML file.

The block elements are available in the `elements` object:

```
def my_title(context):
    return "Chapter 1"
```

```
<h1 tal:content="elements/my_title"></h1>
```

Elements can be used as conditions:

```
def is_footer(context):
    return True
```

```
<footer tal:condition="elements/is_footer">My footer</footer>
```

If an element returns an iterable object (list, dictionary), we can make a loop:

```
def links(context):
    return [
        {'url': 'https://validator.w3.org/', 'title': 'Markup Validation Service'},
        {'url': 'https://www.w3.org/Style/CSS/', 'title': 'CSS'},
    ]
```

```
<ul>
    <li tal:repeat="link elements/links">
        <a tal:attributes="link/url"
           tal:content="link/title"></a>
    </li>
</ul>
```

The current Rapido context is available in the `context` object:

```
<h1 tal:content="context/content/title"></h1>
```

See the TAL commands documentation for more details about TAL.

Create custom views

For now, we have just added small chunks of HTML in existing pages. But Rapido also allows you to create a whole new page (a Plone developer would call it a new **view**).

Let’s imagine we want to display one of our Rapido blocks in the main content area instead of the regular content. We *could* do it with a simple `replace` Diazo rule:

```
<replace css:content="#content">
    <include href="@@rapido/stats/blocks/stats" css:content="form"/>
</replace>
```

But if we do that, the regular content will not be accessible anymore. What if we want to be able to access both the regular content with its regular URL, and define an additional URL to display our block as main content?

To accomplish this, Rapido allows you to declare **neutral views**.

By adding `@@rapido/view/<any-name>` to a content URL we get the content's default view. The `any-name` value can actually be **anything**, we do not really care, we just use it to match a Diazo rule in charge of replacing the default content with our block:

```
<rules if-path="@@rapido/view/show-stats">
    <replace css:content="#content">
        <include css:content="form" href="/@@rapido/stats/blocks/stats" />
    </replace>
</rules>
```

Now if we visit for instance:

```
http://localhost:8080/Plone/page1/@@rapido/view/show-stats
```

we see our block instead of the regular page content.

(And if we visit http://localhost:8080/Plone/page1, we get the regular content of course.)

Exercise 5: Create the Top 5 page

Let's create a block to display the Talks Top 5:

- It needs to be a specific view.
- We will use a TAL template (but for now the content will be fake and static).
- Visitors will access it from a footer link.

Solution

First we create a `top5.pt` file in the `blocks` folder with the following content:

```
<h1 class="documentFirstHeading">Talks Top 5</h1>
<section id="content-core">Empty for now</section>
```

Now we add the following to our `rules.xml` file:

```
<rules if-path="@@rapido/view/talks-top-5">
    <replace css:content-children="#content">
        <include css:content="form" href="/@@rapido/rating/blocks/top5" />
    </replace>
</rules>
```

And then we declare a new action in our footer:

- go to the `site_actions` in the Zope Management Interface:

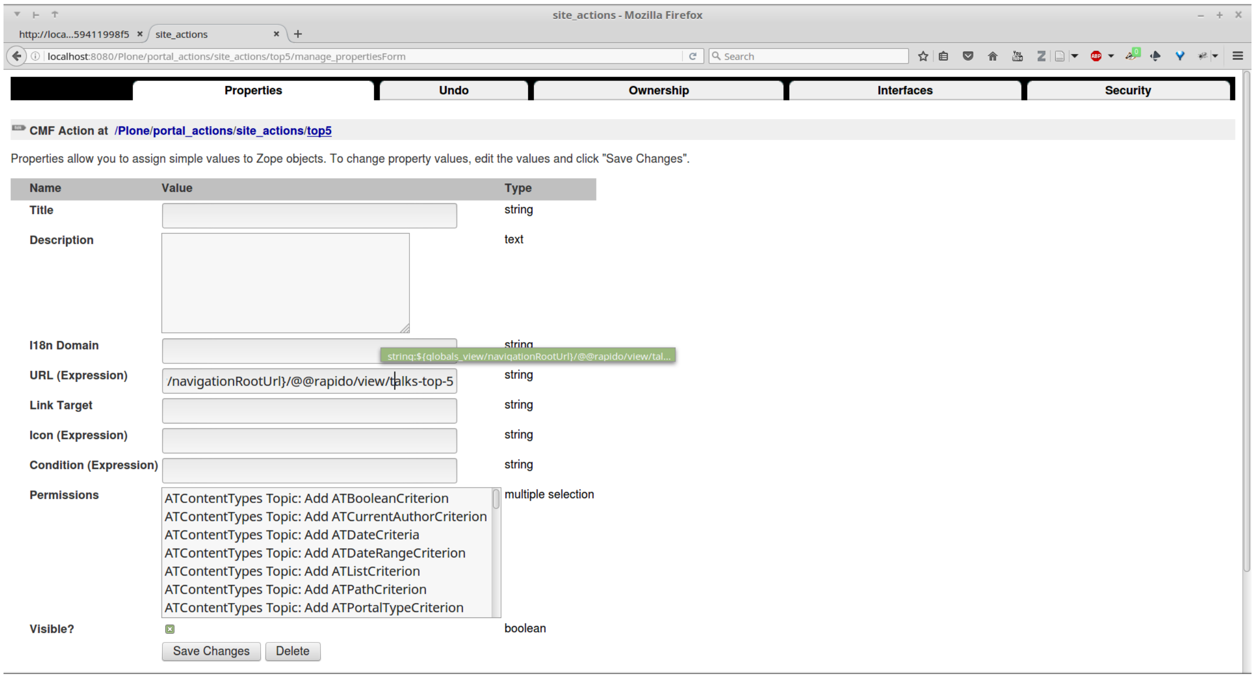
  ```
  http://localhost:8080/Plone/portal_actions/site_actions/manage_workspace
  ```

- add a new `top5` action, with the *URL (Expression)* property set to:

  ```
  string:${globals_view/navigationRootUrl}/@@rapido/view/talks-top-5
  ```

  New in version 5.1: go to *Site Setup -> Actions* add a new action in Site actions category with name “Top 5” and as URL:

2.
```
string:${globals_view/navigationRootUrl}/@@rapido/view/talks-top-5
```


site_actions - Mozilla Firefox
http://loca...59411998f5
site_actions
localhost:8080/Plone/portal_actions/site_actions/top5/manage_propertiesForm
Search
Properties
Undo
Ownership
Interfaces
Security

CMF Action at /Plone/portal_actions/site_actions/top5

Properties allow you to assign simple values to Zope objects. To change property values, edit the values and click "Save Changes".

Name	Value	Type
Title		string
Description		text
I18n Domain		string
URL (Expression)	/navigationRootUrl}/@@rapido/view/talks-top-5	string
Link Target		string
Icon (Expression)		string
Condition (Expression)		string
Permissions	ATContentTypes Topic: Add ATBooleanCriterion ATContentTypes Topic: Add ATCurrentAuthorCriterion ATContentTypes Topic: Add ATDateCriteria ATContentTypes Topic: Add ATDateRangeCriterion ATContentTypes Topic: Add ATListCriterion ATContentTypes Topic: Add ATPathCriterion ATContentTypes Topic: Add ATPortalTypeCriterion	multiple selection
Visible?	☒	boolean

string:${globals_view/navigationRootUrl}/@@rapido/view/tal...

Index and query records

Rapido record items can be indexed, so we can filter or sort records easily.

Indexing is declared in the block YAML file using the `index_type` property. Example:

```
target: ajax
elements:
    firstname:
        type: BASIC
        index_type: field
```

The `index_type` property can have two possible values:

`field` A field index matches exact values, and supports comparison queries, range queries, and sorting.

`text` A text index matches contained words (applicable for text values only).

Queries use the *CQE format* (see documentation.

Example (assuming `author`, `title` and `price` are existing indexes):

```
context.app.search(
    "author == 'Conrad' and 'Lord Jim' in title",
    sort_index="price")
```

To reindex a record, we can use the Rapido Python API:

```
myrecord.save()  # this will also run the on_save event
myrecord.reindex()  # this will just (re-)index the record
```

We can also reindex all the records using the `refresh` URL command:

```
http://myserver.com/Plone/@@rapido/<app-id>/refresh
```

Exercise 6: Compute the top 5

We want to be able to sort the records according to their votes:

- we need to declare `total` as an indexed element,
- we need to refresh all our stored records,
- we need to update the `top5` block to display the first 5 ranked talks.

Solution

We add the following to `rate.yaml` containing:

```
elements:
    ...
    total:
        type: BASIC
        index_type: field
```

To index the previously stored values, we have to refresh the storage index by calling the following URL:

```
http://localhost:8080/Plone/@@rapido/rating/refresh
```

And to make sure future changes will be indexed, we need to fix the `like()` function in the `rate` block: the indexing is triggered when we call the record's `save()` method:

```
def like(context):
    content_path = context.content.absolute_url_path()
    record = context.app.get_record(content_path)
    if not record:
        record = context.app.create_record(id=content_path)
        record['total'] = 0
    record['total'] += 1
    record.save(block_id='rate')
```

Now let's change the `top5` block:

- create `top5.yaml`:

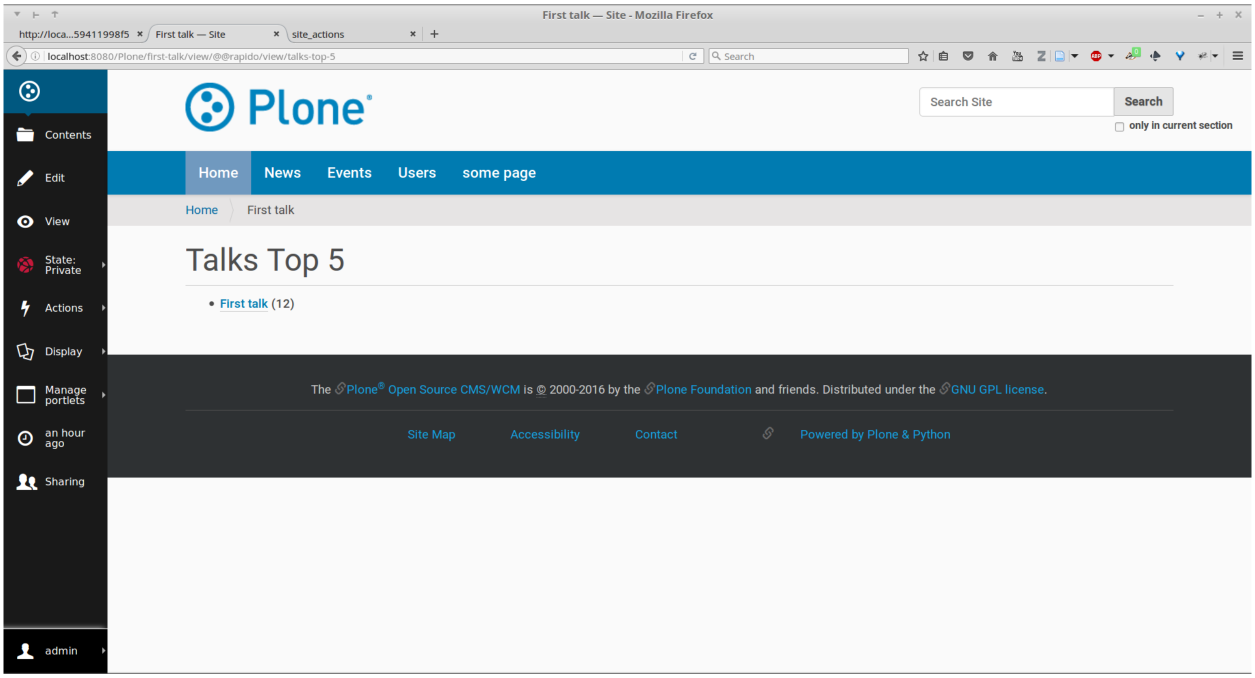
```
elements:
    talks:
        type: BASIC
```

- create `top5.py`:

```
def talks(context):
    search = context.app.search(
        "total>0", sort_index="total", reverse=True)[:5]
    results = []
    for record in search:
        content = context.api.content.get(path=record["id"])
        results.append({
            'url': content.absolute_url(),
            'title': content.title,
            'total': record["total"]
        })
    return results
```

- update `top5.pt`:

```
<h1 class="documentFirstHeading">Talks Top 5</h1>
<section id="content-core">
    <ul>
        <li tal:repeat="talk elements/talks">
            <a tal:attributes="href talk/url"
               tal:content="talk/title">the talk</a>
            (<span tal:content="talk/total">10</span>)
        </li>
    </ul>
</section>
```



Talks Top 5

- First talk (12)

The Plone® Open Source CMS/WCM is © 2000-2016 by the Plone Foundation and friends. Distributed under the GNU GPL license.

Create custom content-rules

Plone content rules allow triggering a given action depending on an *event* (content modified, content created, etc.) and on a *list of criteria* (for example: only for certain content types, only in this folder, etc.).

Plone provides a set of useful ready-to-use content rule actions, such as moving some content somewhere, sending mail to an email address, executing a workflow change, etc.

Rapido allows us to easily implement our own actions. To do this, it adds a generic "Rapido action" to the Plone content rules system. It allows us to enter the following parameters:

- the app id,
- the block id,
- the function name.

The `content` property in the function's `context` allows access to the content targeted by the content rule.

For instance, to transform the content title to uppercase every time we edit something, we would use a function such as this:

```
def upper(context):
    context.content.title = context.content.title.upper()
```

Exercise 7: Reset the votes on workflow change

We would like to reset the votes when we change the workflow status of a talk.

We will need to:

- create a new block to handle our `reset()` function,
- add a content rule to our Plone site,
- assign the rule to the proper location.

Solution

- create `contentrule.py`:

```
def reset(context):
    talk_path = context.content.absolute_url_path()
    record = context.app.get_record(talk_path)
    if record:
        record['total'] = 0
```

- go to *Site setup* -> *Content rules*, and add a rule for the event *State has changed*,
- add a condition on the content type to only target *Talks*,
- add a Rapido action where the application is `rating`, the block is `contentrule` and the method is `reset`,
- activate the rule for the whole site.

Other topics

The following Rapido features haven't been covered by this training:

- using Rapido blocks as tiles in Mosaic,
- using blocks as forms to create, display and edit records directly,
- access control,
- Rapido JSON REST API.

You can find information about those features and also interesting use cases in the Rapido documentation.

Plone Training Solr

Contents:

Set up Plone and Solr

For using Solr with Plone you need two things:

1. A running Solr server
2. An integration product (like collective.solr) for delegation of indexing and searching to the Solr server. In this training we will focus on collective.solr for this purpose.

Bootstrap project:

```
$ mkdir plone-training-solr
$ cd plone-training-solr
$ curl -O https://bootstrap.pypa.io/bootstrap-buildout.py
$ curl -O https://raw.githubusercontent.com/collective/collective.solr/master/solr.cfg
$ curl -o plone5.cfg https://raw.githubusercontent.com/collective/minimalplone5/
↪master/buildout.cfg
$ curl -o solr4.cfg https://raw.githubusercontent.com/collective/collective.solr/
↪master/solr-4.10.x.cfg
```

Create a buildout (*buildout.cfg*) which installs both requirements:

```
[buildout]
extends =
    plone5.cfg
    solr.cfg
    solr4.cfg

[instance]
eggs +=
    collective.solr

[versions]
collective.solr = 6.0a1
collective.recipe.solrinstance = 6.0.0b3
```

Run buildout:

```
$ python2.7 bootstrap-buildout.py
$ bin/buildout
```

Start Plone in foreground mode to see that everything is ok:

```
$ bin/instance fg
```

Start Solr in another terminal in foreground mode

```
$ bin/solr-instance fg
```

Solr Buildout

We assume you are more or less familiar with the Plone buildout, but let's analyze the solr buildout configuration a bit.

First we have two buildout parts in *solr.cfg*

```
[buildout]
parts +=
    solr-download
    solr-instance
```

As the name suggests *solr-download* gets the full Solr package from the official download server and unpacks it. The part *solr-instance* is for configuring Solr. Let's continue with the details.

The base Solr settings specify the host (usually localhost or 127.0.0.1), the port (8983 is the standard port of Solr) and two Java parameters for specifying lower and upper memory limit. More is usually better.

```
[settings]
solr-host = 127.0.0.1
solr-port = 8983
solr-min-ram = 128M
solr-max-ram = 256M
```

If you want a rough idea on how much memory you should use, follow the guidelines found in this article:

See also:

https://lucidworks.com/blog/2011/09/14/estimating-memory-and-storage-for-lucenesolr/

There is nothing fancy in the Solr download part. It takes an URL to the Solr binary and an md5 sum for verification.

Note: At time of writing the latest working version of Solr was 4.10.x

It looks like this in *solr.cfg* and *solr4.cfg*

```
[solr-download]
recipe = hexagonit.recipe.download
strip-top-level-dir = true

[solr-download]
url = https://archive.apache.org/dist/lucene/solr/4.10.4/solr-4.10.4.tgz
md5sum = 8ae107a760b3fc1ec7358a303886ca06
```

The Solr instance part is more complicated. It provides a subset of many, many configuration options of Solr and the possibility to define the schema of the index:

```
[solr-instance]
recipe = collective.recipe.solrinstance
solr-location = ${solr-download:location}
host = ${settings:solr-host}
port = ${settings:solr-port}
basepath = /solr
max-num-results = 500
section-name = SOLR
unique-key = UID
logdir = ${buildout:directory}/var/solr
default-search-field = default
default-operator = and
java_opts =
  -Dcom.sun.management.jmxremote
  -Djava.rmi.server.hostname=127.0.0.1
  -Dcom.sun.management.jmxremote.port=8984
  -Dcom.sun.management.jmxremote.ssl=false
  -Dcom.sun.management.jmxremote.authenticate=false
  -server
  -Xms${settings:solr-min-ram}
  -Xmx${settings:solr-max-ram}
```

Let's analyze them one by one

```
solr-location = ${solr-download:location}
```

Specify the location of Solr, dowloaded with the previous part.

```
host = ${settings:solr-host}
port = ${settings:solr-port}
basepath = /solr
```

Base configuration for running Solr referencing previously defined settings. With this configuration it is possible to access Solr in a browser with the following URL: http://localhost:8983/solr

The section-name defines the name which can be used to reflect custom address and/or basepath settings in zope.conf.:

```
section-name = SOLR
```

It follows the following pattern in *zope.conf*: if you use standard settings no changes in *zope.conf* are necessary.

```
<product-config ${part:section-name}>
    address ${part:host}:${part:port}
    basepath ${part:basepath}
</product-config>
```

Note: Another easy way to use different hosts on dev, stage and production machines is to define a host alias in /etc/hosts

Like the Zope ZCatalog the Solr index has a schema consisting of index and metadata fields. You can think of index fields as something you can use for querying / searching and metadata something you return as result list. Solr defines its schema in a big XML file called `schema.xml`. There is a section in the `collective.recipe.solrinstance` buildout recipe which gives you access to the most common configuration options in a buildout way:

```
index =
    name:allowedRolesAndUsers   type:string stored:false multivalued:true
    name:created                type:date stored:true
    name:Creator                type:string stored:true
    name:Date                   type:date stored:true
    name:default                type:text indexed:true stored:false multivalued:true
    name:Description            type:text copyfield:default stored:true
    name:description            type:text copyfield:default stored:true
    name:effective              type:date stored:true
    name:exclude_from_nav       type:boolean indexed:false stored:true
    name:expires                type:date stored:true
    name:getIcon                type:string indexed:false stored:true
    name:getId                  type:string indexed:false stored:true
    name:getRemoteUrl           type:string indexed:false stored:true
    name:is_folderish           type:boolean stored:true
    name:Language               type:string stored:true
    name:modified               type:date stored:true
    name:object_provides        type:string stored:false multivalued:true
    name:path_depth             type:integer indexed:true stored:false
    name:path_parents           type:string indexed:true stored:false multivalued:true
    name:path_string            type:string indexed:false stored:true
    name:portal_type            type:string stored:true
    name:review_state           type:string stored:true
    name:SearchableText         type:text copyfield:default stored:false
    name:searchwords            type:string stored:false multivalued:true
    name:showinsearch           type:boolean stored:false
    name:Subject                type:string copyfield:default stored:true multivalued:
↪true
    name:Title                  type:text copyfield:default stored:true
    name:Type                   type:string stored:true
    name:UID                    type:string stored:true required:true
```

- name: Name of the field
- type: Type of the field (e.g. `string` , `text`, `date`, `boolean`)
- indexed: The field is searchable
- stored: The field is returned as metadata
- copyfield: copy content to another field, e.g. copy title, description, subject and SearchableText to default.

For a complete list of schema configuration options refer to Solr documentation.

See also:

https://wiki.apache.org/solr/SchemaXml#Common_field_options

This is the bare minimum for configuring Solr. There are more options supported by the buildout recipe `collective.recipe.solrinstance` and even more by Solr itself. Most notably are the custom extensions for *schema.xml* and *solrconfig.xml*. We will see examples for this later on in the training.

Or you can even point to a custom location for the main configuration files.

```
schema-destination = ${buildout:directory}/etc/schema.xml
config-destination = ${buildout:directory}/etc/solrconfig.xml
```

After running the buildout, which downloads and configures Solr and Plone we are ready to fire both servers.

Plone and Solr

To activate Solr in Plone *collective.solr* needs to be activated as an addon in Plone.

Activating the Solr addon adds a configuration page to the controlpanel. It can be accessed via <PORTAL_URL>/@@solr-controlpanel or via “Configuration” -> “Solr Settings”

Check: “Active”, click “Save”

Activating Solr in the controlpanel activates a patch of Plones indexing and search methods to use Solr for indexing and querying.

Note: Note that ZCatalog is not replaced but Solr is *additionally* used for indexing and searching.

Control panel configuration options

- *Active* - Turn connection between Plone and Solr on/off.
- *Host* - The host name of the Solr instance to be used. Defaults to 127.0.0.1
- *Port* - The port of the Solr instance to be used. Defaults to 8983
- *Base* - The base prefix of the Solr instance to be used. Defaults to /solr
- *Asynchronous indexing* - Check to enable asynchronous indexing operations, which will improve Zope response times in return for not having the Solr index updated immediately.
- *Automatic commit* - If enabled each index operation will cause a commit to be sent to Solr, which causes it to update its index. If you disable this, you need to configure commit policies on the Solr server side.
- *Commit within*

Timeouts and search limit

- Index timeout
- Search timeout
- Maximum search results

Search query configuration

- Required query parameters
- Pattern for simple search queries
- Default search facets
- Filter query parameters
- Slow query threshold
- Effective date steps
- Exclude user from allowedRolesAndUsers

Highlighting

https://wiki.apache.org/solr/HighlightingParameters

- Highlighting fields
- Highlight formatter: pre
- Highlight formatter: post
- Highlight Fragment Size
- Default fields to be returned
- Levensthein distance

Atomic updates and boosting

- Enable atomic updates
- Python script for custom index boosting

With Solr activated, searching in Plone works like the following:

- Search contains one of the fields set as required (which is normally the fulltext field *SearchableText*) -> Solr results are returned
- Search does not contain all fields marked as required -> ZCatalog results are returned. Which is the case for rendering the navigation, folder contents, etc.
- The search contains the stanza *use_solr=True*. -> Solr results are returned independent of the required fields.

Then you are ready for your first search. Search for *Plone*. You should get the frontpage as a result–which is not super awesome at the first place because we have this without Solr too–but it is the first step in utilizing the full power of Solr.

Configuration with ZCML

Another way to configure the connection is via ZCML. You can use the following snippet to configure host, port und basepath:

```
<configure xmlns:solr="http://namespaces.plone.org/solr">
  <solr:connection host="127.0.0.23" port="3898" base="/foo" />
</configure>
```

The ZCML configuration takes predence over the configuration in the registry / control-panel.

Committing strategies

Synchronous immediately

The default commit strategy is to commit to Solr on every Zope commit. This ensures an always up to date index but may come at cost of indexing time especially when doing batch operations like data import.

To use this behavior, turn **Automatic commit** ON in the Solr controlpanel in Plone.

Synchronous batched

Another commit strategy is to do timed commits in Solr. This method is usually way faster but comes with the cost of index delays.

To use this behavior you have to do two things:

- Turn **Automatic commit** OFF in the Solr controlpanel in Plone.
- Set one or both of the following options in the Solr server configuration via the collective.recipe.solrinstance buildout recipe:
  - `autoCommitMaxDocs` - The number of updates that have occurred since the last commit.
  - `autoCommitMaxTime` - The number of milliseconds since the oldest uncommitted update.

Asynchronous

The third commit stragey is to do full asynchronous commits. This can be activated by setting the Flag **Asynchronous indexing** in the Solr control panel to ON. This behavior is the most efficient in terms of Zope response time. Since it is fire and forget the consistency could be harmed in midterm. It is advisable to to a sync or full-index from time to time if you work with this strategy.

Additional information can be found in the Solr documentation:

See also:

https://cwiki.apache.org/confluence/display/solr/UpdateHandlers+in+SolrConfig#UpdateHandlersinSolrConfig-commitWithin

Excercise

Have a running Plone and Solr with collective.solr active and experiment with commit strategies.

Solr GUI and Query Syntax

In the next part we will take a closer look the the search GUI of Solr and its query syntax.

Access Solr GUI

Solr is a REST-based wrapper around the Java lucene index. It comes with its own web GUI. It is possible to access all of the SOLR API via REST and most of this functionality is exposed via its web GUI. To test it out, do the following:

- Go to: http://localhost:8983/solr/#/
- Select Core "collection1"
- Go to: "Schema Browser"
- Select "fullname"
- Click: "Load Term Info"
- Click on term "<fullname>"

Solr Query Syntax

Solr Query Parameters:

Query "q":

```
Title:"news"
*:"news"
```

Solr response

```
{
  "responseHeader":{
    "status":0,
    "QTime":0,
    "params":{
      "q":"*:*",
      "indent":"true",
      "wt":"json"}},
  "response":{"numFound":51,"start":0,"docs":[
      {
        "path_string":"/Plone/news",
        "Title":"News",
        "showinsearch":true,
        "path_depth":3,
        "exclude_from_nav":false,
        "Type":"Folder",
        "UID":"88411960ec3f4b1f86feae9094ba718e",
        "is_folderish":true,
        "getId":"news",
        "Date":"2015-12-25T16:46:24Z",
        "review_state":"published",
        "Language":"en",
        "portal_type":"Folder",
        "expires":"2499-12-30T22:00:00Z",
        "allowedRolesAndUsers":["Anonymous"],
        "path_parents":["/Plone",
          "/Plone/news"],
        "object_provides":["Products.ATContentTypes.interfaces.folder.IATFolder",
          "Products.CMFPlone.interfaces.syndication.ISyndicatable",
          "eea.facetednavigation.subtypes.interfaces.IPossibleFacetedNavigable",
          "Products.CMFCore.interfaces._content.IContentish",
          "webdav.interfaces.IWriteLock"],
        "Description":"Site News",
        "effective":"1000-01-05T22:00:00Z",
        "created":"2015-12-25T16:46:24.841Z",
        "getIcon":"",
        "Creator":"admin",
        "modified":"2015-12-25T16:46:24.841Z",
        "SearchableText":"news  News  Site News ",
        "_version_":1545835799688249344},
```

Filter Query "fq":

This parameter can be used to specify a query that can be used to restrict the super set of documents that can be returned, without influencing the score. It can be very useful for speeding up complex queries since the queries specified with fq are cached independently from the main query. Caching means the same filter is used again for a later query (i.e. there's a cache hit). See SolrCaching to learn about the caches Solr uses:

```
is_folderish:true
```

Sorting "sort":

```
"Date asc"
"Date desc"
```

Filter List "fl":

```
Title,Type
```

This parameter can be used to specify a set of fields to return, limiting the amount of information in the response.

Response Writer "wt":

```
"json"
```

A Response Writer generates the formatted response of a search.

Solr Query via URL

Copy query from Solr GUI, e.g.:

```
http://localhost:8983/solr/collection1/select?q=Title%3A%22termine%22&wt=json&
↪indent=true
```

You can use curl or the Python package *requests* (https://pypi.python.org/pypi/requests) to access the REST API of Solr.

Solr Query via API

Another way of accessing Solr is to use a Python wrapper, which exposes the Solr API in a Pythonic way. Collective.solr has included such a wrapper (`solr.py`), which is old but still works for our case. Meanwhile there are other packages around. Here are some examples:

- `mysolr`: https://pypi.python.org/pypi/mysolr/0.8.3
- `solrpy`: https://pypi.python.org/pypi/solrpy3/0.98
- `pysolr`: https://pypi.python.org/pypi/pysolr/3.5.0

Sometimes it is handy to have a separate virtualenv available for doing batch operations (delete, update, etc.)

I use the following script to delete all Plone Documents from Solr

```
>>> from mysolr import Solr
>>> solr = Solr(base_url='http://localhost:8983/solr')
>>> solr.delete_by_query('portal_type:Document')
```

Advanced Solr Query Syntax

Simple Query:

```
"fieldname:value"
```

A clause can be **mandatory** (finds only articles containing the word *Boston*):

```
+Boston
```

A clause can be **probibited** (finds all articles except those containing the word *Vienna*):

```
-Vienna
```

Operators:

```
"Title:Foo AND Description:Bar"
```

“AND”, “OR”, “+”, “-”, “||”, “NOT”

Be carefull with combining operators such as:

```
New AND York OR Buenos AND Aires
```

which will probably lead to no results. You will need to use sub-queries.

Sub-queries:

```
(New AND York) OR (Buenos Aires)
```

Range Queries:

```
"[* TO NOW]"
```

Boost Terms:

```
"people^4"
```

Fuzzy Search:

```
"house0.6"
```

Proximity Search:

```
"apache solr"~
```

with treshold:

```
"apache solr"~7
```

Wildcard queries:

Find all cities starting with *New* you can do:

```
New*
```

Or a single character wildcard:

```
M?ller
```

which will find *Müller*, *Miller*, etc.

Date math

Solr provides some useful date units which are available for date queries. The units you can choose of are:

YEAR, *MONTH*, *DAY*, *DATE* (synonymous with *DAY*), *HOUR*, *MINUTE*, *SECOND*, *MILLISECOND*, *MILLI* (synonymous with *MILLISECOND*) and *NOW*. All of these units can be pluralized with an *S* as in *DAYS*.

```
effective:[* TO NOW-3MONTHS]
```

NOW has a millisecond precision. To round down by using the */* operator (it never rounds up):

```
effective:[* TO NOW/DAY-2YEAR]
```

Existing (and non-existing) queries

Assume we want to find all documents which have a value in a certain field (whatever that value is, it doesn't matter).

Find all documents with a description:

```
Description:[* TO *]
```

The oposite (finding all documents with no description) is also possible:

```
-Description:[* TO *]
```

Faceting

Faceting is one of the killer features of Solr. It allows the grouping nd filtering results for better findability. To enable faceting you need o turn faceting on in the query and specify the fields you want tofacet upon:

For a simple facet query in Solr you activate the feature and specify the facet fields(s):

```
http://localhost:8983/solr/collection1/select?q=*%3A*&wt=json&indent=true&facet=true&
↪facet.field=portal_type
```

Besides the matching documents this will give you an additional grouping of documents:

```
{
 "responseHeader":{
  "status":0,
  "QTime":6,
  "params":{
    "q":"*:*",
    "facet.field":"portal_type",
    "indent":"true",
    "wt":"json",
    "facet":"true"}},
 "response":{"numFound":6,"start":0,"docs":[
   ...
 ]}
 "facet_counts":{
  "facet_queries":{},
  "facet_fields":{
    "portal_type":[
      "Folder",3,
      "Collection",2,
      "Document",1]},
  "facet_dates":{},
  "facet_ranges":{},
  "facet_intervals":{}}
}
```

There are more complex scenarios possible. For a complete list of options see the according Solr documentation.

See also:

https://cwiki.apache.org/confluence/display/solr/Faceting

With collective.solr you don't have to worry about the faceting details too much. There is a convenient method to configure the faceting fields in the control panel of collective.solr. All the other magic is handled by the product. We will see an example later.

Search GUIs

- collective.solr out of the box: collective.solr commes with its own search view. For the new version 6.0 it is based on ReactJS and looks similar to the Plone search view with native facet support of Solr.
- eea.facetednavigation: This addon allows faceting out of the box even without Solr. It is a product for integrators to setup search and filter GUIs TTW. It can be used for several use cases: Search pages, collection replacements, etc. **DEMO**

- custom: Another way is to create a custom search page. This is easy to do and we will see later on in this training how.

Exercise

Do some queries in Solr directly

First Steps

Maintenance Task

All the maintenance tasks are accessible through the Solr controlpanel in Plone since version 6.0 of collective.solr. Nevertheless it is good to know the direct URLs sometimes. Another goodie of accessing the URLs directly is they support GET parameters to limit and change their behavior.

Let's see some examples:

Reindex

Reindex all Plone objects found in catalog:

http://localhost:8080/Plone/@@solr-maintainance/reindex

The call of this URL finds all contentish objects (meaning all objects derived from one of the catalog mixin classes) and (re)indexes them.

There are some parameters you can specify:

- *batch* (default:1000): Batch size for commit. Data is only send to Solr on commit.
- *skip* (default:0): Skip N elements when iterating over all contentish objects.
- *limit* (default:0): Only index N elements.
- *ignore_portal_types* (default:None): Blacklist of portal types not to be indexed.
- *only_portal_types* (default:None): Whiltelist of portal types not to be indexed.
- *idxs* (default:[]): Only this index fields will be updated.

Cleanup

Remove entries from Solr that don't have a corresponding Zope object or have a different UID than the real object:

http://localhost:8080/Plone/@@solr-maintainance/cleanup

The only parameter you can specify is the batch size:

- *batch* (default:1000): Batch size for commit. Data is only send to Solr on commit.

Sync Solr Index

Sync the Solr index with the portal catalog. Records contained in the catalog but not in Solr will be indexed and records not contained in the catalog will be removed.

http://localhost:8080/Plone/@@solr-maintainance/sync

There are some parameters you can specify:

- *batch* (default:1000): Batch size for commit. Data is only send to Solr on commit.
- *preImportDeleteQuery* (default::): This **delete** query will be executed on Solr before the sync process starts.

Purge Solr Index

Clear **all** elements from the Solr default collection.

http://localhost:8080/Plone/@@solr-maintainance/clear

There are no parameters you can specify for the clear action.

Note: Be careful with required fields. If you specify required fields in your schema, which are not present in your indexing record indexing will not happen.

Indexing a new dexterity field

A common use case is to add an additional field to the index. We have to inform both sides (Solr and Plone) if we need a new field in the index.

A simple use case is to pass through a raw dexterity field to the index. First we add the field to the schema. We do this TTW right now.

Note: In the production setup you will define your schema with an interface or a supermodel XML but this is beyond of this training. More information on dexterity schemas and fields can be found in the Plone documentation: http://docs.plone.org/external/plone.app.dexterity/docs/schema-driven-types.html

Let's add a field *email* to a task. We assume this is contact email which can be used to contact the responsible support person for this task. And we want to make this field to be found in fulltext search.

It does not matter if we add the field TTW, via supermodel or via interface. The only thing you have to make sure the **name** of the field is identical in Plone and Solr.

Next thing we do is to extend the Solr fields definition in our buildout.cfg.

On the *fields* section of the *solr* part we add the following line:

```
name:email    type:string copyfield:SearchableText stored:true multivalued:false
name:fullname type:string copyfield:SearchableText stored:true multivalued:false
```

After we have done that we need to rerun buildout

```
$ bin/buildout
```

and restart Solr and Plone

```
$ bin/instance restart
$ bin/solr-instance fg
```

This method works out of the box, if the name of the Dexterity field in Plone is the same as the field in the schema of Solr. And assuming you *have* the information you need for the index available as a Dexterity field.

Let's assume we have a field *fullname* in Solr and in Plone we have separate fields for *firstname* and *surname*. We need an indexing adapter to have the fullname indexed. This is done like this:

First we need an indexer binding to our dexterity content:

```
from plone.indexer import indexer
from plonetraining.solr_example.interfaces import ITask

@indexer(ITask)
def fullname_indexer(obj):
    """ Construct a fullname for Solr from Dexterity fields """
    return getattr(obj, 'firstname', '') + ' ' + getattr(obj, 'surname', '')
```

And we need a named adapter, which correlates with the name of the field in Solr (*fullname* in our case).

```
<adapter factory=".indexer.fullname_indexer" name="fullname" />
```

That's it. After adding a new Task or reindexing an existing one with firstname and surname set, the *fullname* in Solr appears.

Note: Pro tip: If you need to modify or extend the existing fulltext implementation in Plone (This could be adding a custom field to it, or remove title or description from it), there is a handy addon for this purpose. It is well documented but further investigation is out of the scope of this training, see https://pypi.python.org/pypi/collective.dexteritytextindexer

Boosting

In a standard installation of Solr all fields are treated equally important for searching. Usually this is not what we want. We want the Title to be more important, or a special type (e.g. News) to be prioritized. Solr offers boosting values at index and at search time. The search boosting is utilized automatically when you install collective.solr. It is configured in the control-panel with the default search pattern:

```
+(Title:{value}^5 OR Description:{value}^2 OR SearchableText:{value} OR
SearchableText:({base_value}) OR searchwords:({base_value})^1000)
```

This reads like this. If a term occurs in the *Title* field prioritize it 5 times, if it is in the *Description* field prioritize it two times. Search but don't prioritize terms occuring in the *SearchableText* index. If a term occurs in the *searchwords* priotize it by value 1000 so it will show always at the top.

You can override this pattern to fit your needs.

Another way to boost documents is at indexing time. For this purpose you can specify a Restricted Python script in Solr control panel. Let's assume we want to put a special emphasis on News Items. Our script looks as follows:

```
return {'': 20} if data.get('portal_type') == 'News Item' else {}
```

This will boost all fields of *News Items* by factor 20. Which means *News Items* will be prioritized in the ranking and show as first search results with the same term.

Note: Boosting at index time is only available if you turn off atomic updates.

Exercise

1. Create or enhance a Dexterity type with an additional field which is indexed.
2. Create a custom indexer in Plone.

How does collective.solr work

Currently we depend on collective.indexing as a means to hook into the normal catalog machinery of Plone to detect content changes. collective.indexing before version two had some persistent data structures that frequently caused problems when removing the add-on. These problems have been fixed in version two. Unfortunately collective.indexing still has to hook the catalog machinery in various evil ways, as the machinery lacks the required hooks for its use-case. Going forward it is expected for collective.indexing to be merged into the underlying ZCatalog implementation, at which point collective.solr can use those hooks directly.

Base Functionality

- Patches the ZCatalog
- Some queries are faster in Solr some are not
- Indexes and Metadata duplicated
- Full text search with SearchableText

Transactions

Solr is not transaction-aware and does not support any kind of rollback or undo. We therefore only send data to Solr at the end of any successful request. This is done via collective.indexing, a transaction manager and an end request transaction hook. This means you won't see any changes done to content inside a request when doing Solr searches later on in the same request.

Querying Solr with collective.solr

ZCatalog Query:

```
catalog(SearchableText='Foo', portal_type='Document')
```

Result is a Solr Object.

Direct Solr Queries:

```
solr_search = solrSearchResults(
    SearchableText=SearchableText,
    spellcheck='true',
    use_solr='true',
)
```

You can pass Solr query params directly to Solr and force a Solr response with

```
use_solr='true'
```

Mangler

collective.solr has a mangleQuery function that translates / mangles ZCatalog query parameters to replace zope specifics with equivalent constructs for Solr.

See also:

https://github.com/collective/collective.solr/blob/master/src/collective/solr/mangler.py#L96

Solr Buildout Configuration

Solr Multi Core

solr.cfg:

```
[solr-instance]
recipe = collective.recipe.solrinstance:mc
cores =
  collection1
  collection2
  collection3
  testing
default-core-name = collection1
```

Note: collective.solr does not support multicore setups currently. It always uses the default core for indexing and searching.

Stopwords

For indexes with lot of text, common uninteresting words like *"the"*, *"a"*, and so on, make the index large and slow down phrase queries. To deal with this problem, it is best to remove them from fields where they show up often.

We need to add the **StopFilterFactory** with a reference to a text file with one stopword per line to the Solr configuration:

solr.cfg:

```
[solr-instance]
recipe = collective.recipe.solrinstance
filter =
    text solr.StopFilterFactory ignoreCase="true" words="${buildout:directory}/etc/
↪stopwords.txt"
java_opts +=
    -Dsolr.allow.unsafe.resourceloading=true
```

Since we don't copy over the stopwords file to the *parts/solr-instance* directory we need to allow Solr reading resource files outside its home directory.

stopwords.txt:

```
a
the
i
```

For some common language secific examples see the Solr git repository:

See also:

https://github.com/apache/lucene-solr/tree/master/lucene/analysis/common/src/resources/org/apache/lucene/analysis/snowball

Stemming

Stemming is a language specific operation which try to reduce terms to a base form.

Here is an example:

```
"riding", "rides", "horses" ==> "ride", "ride", "hors".
```

This can help in some situations but may hurt in others.

For example, if you run an intranet and people usally know exactly what they are looking for it is probably not a good idea, but if you provide a Google-like search where you browse more than search then stemming is probably for you.

If you are interested in this feature look at the Solr documentation here:

See also:

https://wiki.apache.org/solr/LanguageAnalysis

A short example to include a german stemming factory into the buildout is here:

solr.cfg:

```
[solr-instance]
recipe = collective.recipe.solrinstance
...
filter =
#    text solr.GermanMinimalStemFilterFactory  # Less aggressive
#    text solr.GermanLightStemFilterFactory  # Moderately aggressiv
#    text solr.SnowballPorterFilterFactory language="German2"  # More aggressive
    text solr.StemmerOverrideFilterFactory dictionary="${buildout:directory}/etc/
↪stemdict.txt" ignoreCase="false"
java_opts +=
    -Dsolr.allow.unsafe.resourceloading=true
```

stemdict.txt:

```
# english stemming
monkeys monkey
otters  otter

# some crazy ones that a stemmer would never do
dogs    cat

# german stemming
gelaufen    lauf
lief        lauf
risiken     risiko
```

Synonyms

Solr can deal with synonyms. Maybe you run a shop for selling smartphones and you want people typing "iphone", "i-phone" or even "ephone", "ifone", or "iphnoe" to get the latest "iPhone" offers.

A simple synonym like solution is to use the *searchwords* extension which is provided by collective.solr. It is a schemaextender for all types and allows to specify terms which are boosted by factor 1000 in the default search query. For "real" synonyms implemented in Solr you can use the *SynonymFilterFactory*:

solr.cfg:

```
[solr]
recipe = collective.recipe.solrinstance
...
filter-index =
# The recommended approach for dealing with synonyms is to expand the synonym
# when indexing. See: http://wiki.apache.org/solr/AnalyzersTokenizersTokenFilters
↪#solr.SynonymFilterFactory
    text solr.SynonymFilterFactory synonyms="${buildout:directory}/etc/synonyms.txt"␣
↪ignoreCase="true" expand="true"
```

Note that the SynonymFilterFactory is an index filter and not a query filter.

synonyms.txt:

```
#Explicit mappings match any token sequence on the LHS of "=>"
#and replace with all alternatives on the RHS.  These types of mappings #ignore the␣
↪expand parameter in the schema.
#Examples:
ipod => i-pod, i pod => ipod,

#Equivalent synonyms may be separated with commas and give no explicit mapping.
# In this case the mapping behavior will be taken from the expand parameter in the␣
↪schema.
# This allows the same synonym file to be used in different synonym handling␣
↪strategies.
#Examples:
ipod, i-pod, i pod
foozball , foosball
universe , cosmos

# expand: (optional; default: true) If true, a synonym will be expanded to all
# equivalent synonyms. If false, all equivalent synonyms will be reduced
# to the first in the list.

#multiple synonym mapping entries are merged.
foo => foo bar
foo => baz
#is equivalent to
foo => foo bar, baz
```

For a full list of index and query filter factories consult the Solr documentation:

See also:

https://cwiki.apache.org/confluence/display/solr/Understanding+Analyzers%2C+Tokenizers%2C+and+Filters

Exercise

Experiment with stemming, stopwords and synonyms. Add your own values and see how Solr behaves.

More Features...

Next we will cover some more advanced topics which need configuration on Plone and Solr side. Features like autocomplete and suggest (did you mean ...) are often requested when it comes to search. They are perfectly doable with the Plone / Solr combination. At the end of this chapter we will build a full search page with autocomplete, suggest, term highlighting and faceting turned on.

Let’s see how and start with autocomplete:

Autocomplete

For autocomplete we need a special Solr handler because we don’t search full terms but only part of terms.

With the additional Solr configuration autocomplete can be called via URL directly:

```
http://localhost:8080/Plone/@@solr-autocomplete?term=Pl
```

Which gives the response

```
[
    {
        "value": "Willkommen bei Plone",
        "label": "Willkommen bei Plone"
    }
]
```

solr.cfg:

```
[solr-instance]
recipe = collective.recipe.solrinstance
...
name:title_autocomplete type:text_auto indexed:true stored:true
name:description_autocomplete type:text_desc indexed:true stored:true

additional-solrconfig =
  <!-- request handler to return typeahead suggestions -->
  <requestHandler name="/autocomplete" class="solr.SearchHandler">
    <lst name="defaults">
      <str name="echoParams">explicit</str>
      <str name="defType">edismax</str>
      <str name="rows">10</str>
      <str name="fl">description_autocomplete,title_autocomplete,score</str>
      <str name="qf">title_autocomplete^30 description_autocomplete^50.0</str>
      <str name="pf">title_autocomplete^30 description_autocomplete^50.0</str>
      <str name="group">true</str>
      <str name="group.field">title_autocomplete</str>
      <str name="group.field">description_autocomplete</str>
      <str name="sort">score desc</str>
      <str name="group.sort">score desc</str>
    </lst>
  </requestHandler>
```

```
extra-field-types =
  <fieldType class="solr.TextField" name="text_auto">
    <analyzer>
      <tokenizer class="solr.WhitespaceTokenizerFactory"/>
      <filter class="solr.LowerCaseFilterFactory"/>
      <filter class="solr.ShingleFilterFactory" maxShingleSize="4" outputUnigrams=
↪"true"/>
      <filter class="solr.EdgeNGramFilterFactory" maxGramSize="20" minGramSize="1"/>
     </analyzer>
  </fieldType>
  <fieldType class="solr.TextField" name="text_desc">
    <analyzer>
      <tokenizer class="solr.WhitespaceTokenizerFactory"/>
      <filter class="solr.LowerCaseFilterFactory"/>
      <filter class="solr.ShingleFilterFactory" maxShingleSize="4" outputUnigrams=
↪"true"/>
      <filter class="solr.EdgeNGramFilterFactory" maxGramSize="20" minGramSize="1"/>
     </analyzer>
   </fieldType>

additional-schema-config =
  <copyField source="Title" dest="title_autocomplete" />
  <copyField source="Description" dest="description_autocomplete" />
```

For the search template we utilize the HTML5 datalist element to populate the search input field.

search.pt:

```
<html lang="en"
      metal:use-macro="context/main_template/macros/master"
      i18n:domain="plone">
<body>
  <metal:content-core fill-slot="content-core">
    <input type="text" list="searchresults"
           id="acsearch" placeholder="Search site ..." />
    <datalist id="searchresults" />

    <script>
      $(document).ready(function() {
        $("#acsearch").on("input", function(e) {
          var val = $(this).val();
          if(val.length < 2) return;
          $.get("solr-autocomplete", {term:val}, function(res) {
            var dataList = $("#searchresults");
            dataList.empty();
            if(res.length) {
              for(var i=0, len=res.length; i<len; i++) {
                var opt = $("<option></option>").attr("value", res[i].label);
                dataList.append(opt);
              }
            }
          }, "json");
        });
      })
    </script>
  </metal:content-core>
</body>
</html>
```

Suggest

The suggest (did you mean ...) feature is well known from popular search engines. It is integrated into Solr as a component which needs to be enabled and configured. Here is an example configuration which works with collective.solr. If you change it stick to the names of the parameters and handlers.

The JSON view of Plone can be called with this URL:

```
http://localhost:8080/Plone/@@search?format=json&SearchableText=Plane
```

And from JavaScript:

```
GET http://localhost:8080/Plone/@@search?SearchableText=Plane
Accept: application/json
```

We get a response like this:

```
{
    "data": [ ],
    "suggestions":
    {
        "plane":
        {
            "endOffset": 87,
            "numFound": 1,
            "startOffset": 82,
            "suggestion":
                [
                    "plone"
                ]
            }
        }
    }
}
```

The configuration in buildout is as follows:

```
[solr-instance]
recipe = collective.recipe.solrinstance
...

additional-solrconfig =
  <!-- =================================================================== -->
  <!-- SUGGEST                                                             -->
  <!-- =================================================================== -->
   <!-- Spell Check

        The spell check component can return a list of alternative spelling
        suggestions.

        http://wiki.apache.org/solr/SpellCheckComponent
     -->
  <searchComponent name="spellcheck" class="solr.SpellCheckComponent">

    <str name="queryAnalyzerFieldType">SearchableText</str>

    <!-- Multiple "Spell Checkers" can be declared and used by this
         component
      -->
    <!-- a spellchecker built from a field of the main index -->
    <lst name="spellchecker">
      <str name="name">default</str>
      <str name="field">SearchableText</str>
      <str name="classname">solr.DirectSolrSpellChecker</str>
      <!-- the spellcheck distance measure used, the default is the internal␣
↪levenshtein -->
      <str name="distanceMeasure">internal</str>
      <!-- minimum accuracy needed to be considered a valid spellcheck suggestion -->
      <float name="accuracy">0.5</float>
      <!-- the maximum #edits we consider when enumerating terms: can be 1 or 2 -->
      <int name="maxEdits">2</int>
      <!-- the minimum shared prefix when enumerating terms -->
      <int name="minPrefix">1</int>
      <!-- maximum number of inspections per result. -->
      <int name="maxInspections">5</int>
      <!-- minimum length of a query term to be considered for correction -->
      <int name="minQueryLength">4</int>
      <!-- maximum threshold of documents a query term can appear to be considered␣
↪for correction -->
      <float name="maxQueryFrequency">0.01</float>
      <!-- uncomment this to require suggestions to occur in 1% of the documents
        <float name="thresholdTokenFrequency">.01</float>
      -->
    </lst>

    <!-- a spellchecker that can break or combine words.  See "/spell" handler below␣
↪for usage -->
    <lst name="spellchecker">
      <str name="name">wordbreak</str>
      <str name="classname">solr.WordBreakSolrSpellChecker</str>
      <str name="field">SearchableText</str>
      <str name="combineWords">true</str>
      <str name="breakWords">true</str>
      <int name="maxChanges">10</int>
    </lst>

    <!-- Custom Spellchecker -->
    <lst name="spellchecker">
      <str name="name">suggest</str>
      <str name="classname">org.apache.solr.spelling.suggest.Suggester</str>
      <str name="lookupImpl">org.apache.solr.spelling.suggest.fst.WFSTLookupFactory</
↪str>
      <str name="field">SearchableText</str>
      <float name="threshold">0.0005</float>
      <str name="buildOnCommit">true</str>
    </lst>

  </searchComponent>

  <!-- A request handler for demonstrating the spellcheck component.

       NOTE: This is purely as an example.  The whole purpose of the
       SpellCheckComponent is to hook it into the request handler that
       handles your normal user queries so that a separate request is
       not needed to get suggestions.
       IN OTHER WORDS, THERE IS REALLY GOOD CHANCE THE SETUP BELOW IS
       NOT WHAT YOU WANT FOR YOUR PRODUCTION SYSTEM!

       See http://wiki.apache.org/solr/SpellCheckComponent for details
       on the request parameters.
    -->
  <requestHandler name="/spell" class="solr.SearchHandler" startup="lazy">
    <lst name="defaults">
      <!-- Solr will use suggestions from both the 'default' spellchecker
           and from the 'wordbreak' spellchecker and combine them.
           collations (re-written queries) can include a combination of
           corrections from both spellcheckers -->
      <str name="spellcheck.dictionary">default</str>
      <str name="spellcheck.dictionary">wordbreak</str>
      <str name="spellcheck.dictionary">suggest</str>
      <str name="spellcheck">on</str>
      <str name="spellcheck.extendedResults">true</str>
      <str name="spellcheck.count">10</str>
      <str name="spellcheck.alternativeTermCount">5</str>
      <str name="spellcheck.maxResultsForSuggest">5</str>
      <str name="spellcheck.collate">true</str>
      <str name="spellcheck.collateExtendedResults">true</str>
      <str name="spellcheck.maxCollationTries">10</str>
      <str name="spellcheck.maxCollations">5</str>
    </lst>
    <arr name="last-components">
      <str>spellcheck</str>
    </arr>
  </requestHandler>
```

A simple integration in our training-search is here:

```
<html lang="en"
      metal:use-macro="context/main_template/macros/master"
      i18n:domain="plone">
<body>
  <metal:content-core fill-slot="content-core">
    <input type="text" list="searchresults"
           id="acsearch" placeholder="Search site ..." />
    <datalist id="searchresults" />

    <script>
      $(document).ready(function() {
        $("#acsearch").on("input", function(e) {
          var val = $(this).val();
          if(val.length < 2) return;
          $.get("suggest-terms", {term:val}, function(res) {
            var dataList = $("#searchresults");
            dataList.empty();
            if(res.length) {
              for(var i=0, len=res.length; i<len; i++) {
                var opt = $("<option></option>").attr("value", res[i].label);
                dataList.append(opt);
              }
            }
          }, "json");
        });
      })
    </script>
  </metal:content-core>
</body>
</html>
```

Facetting

Facetting is tightly integrated in `collective.solr` and works out of the box. We will now create a full search page with faceting, autocomplete, search term highlighting and suggest enabled. The HTML of the page is mainly taken from the standard page. To reduce complexity some of the standard features like syndication, i18n and view actions has been removed:

```
<html metal:use-macro="here/main_template/macros/master">
<head>
  <metal:block fill-slot="top_slot"
               tal:define="dummy python:request.set('disable_border',1);
                   disable_column_one python:request.set('disable_plone.leftcolumn
↪',1);
                   enable_column_two python:request.set('disable_plone.rightcolumn
↪',0);"/>
  <metal:block fill-slot="column_one_slot"/>

  <metal:js fill-slot="javascript_head_slot">
    <script type="text/javascript" src=""
            tal:attributes="src string:${portal_url}/++resource++collective.showmore.
↪js">
    </script>
    <script type="text/javascript">

  $(document).ready(function() {
    $("#acsearch").on("input", function(e) {
      var val = $(this).val();
      if(val.length < 2) return;
      $.get("solr-autocomplete", {term:val}, function(res) {
        var dataList = $("#searchresults");
        dataList.empty();
        if(res.length) {
          for(var i=0, len=res.length; i<len; i++) {
            var opt = $("<option></option>").attr("value", res[i].label);
            dataList.append(opt);
          }
        }
      },"json");
    });
  })


    </script>
  </metal:js>
</head>

<body>
<div metal:fill-slot="main"
     tal:define="results view/search">
  <form name="searchform"
        action="search"
        class="searchPage"
        tal:attributes="action request/getURL">
    <input class="searchPage" name="SearchableText" id="acsearch" type="text"
           size="25" list="searchresults" title="Search Site"
           placeholder="Search Site ..."
           tal:attributes="value request/SearchableText|nothing;"/>
    <datalist id="searchresults"/>
    <input class="searchPage searchButton" type="submit" value="Search"/>
    <div tal:define="view nocall: context/@@search-facets | nothing"
         tal:condition="python: view"
         tal:replace="structure view/hiddenfields"/>
  </form>
  <h1 class="documentFirstHeading">
    Search results
    <span class="discreet">
        &mdash;
      <span tal:content="python:len(results)">234</span>
      items matching your search terms
    </span>
  </h1>

  <div tal:condition="not: view/has_results">
    <p tal:define="suggest view/suggest">
      <tal:noresuls>No results were found.</tal:noresuls>
      <tal:suggest condition="suggest">Did you mean:
        <strong>
          <a href="" tal:attributes="href suggest/url"
             tal:content="suggest/word">Plone</a>
        </strong>
      </tal:suggest>
    </p>
  </div>
  <div tal:condition="results" id="content-core">
    <dl class="searchResults">
      <tal:results repeat="result results">
        <dt tal:attributes="class result/ContentTypeClass">
          <a href="#"
             tal:attributes="href result/getURL;
                             class string:state-${result/review_state}"
             tal:content="result/Title"/>
        </dt>
        <dd>
          <span tal:replace="result/CroppedDescription">Cropped description</span>
          <br/>
        </dd>
      </tal:results>
    </dl>
    <div metal:use-macro="here/batch_macros/macros/navigation"/>
  </div>

</div>
<div metal:fill-slot="portlets_two_slot">
  <div tal:define="facet_view nocall: context/@@search-facets;
                   results view/search;"
       tal:condition="view/has_results"
       tal:replace="structure python:facet_view(results=results._sequence._
↪basesequence)"/>
</div>
```

```
</body>
</html>
```

Let's analyze the important parts. The head includes a reference to the `showmore.js` JavaScript, which is included in `collective.solr` and used to reduce long lists of facets. Additionally the left column is removed on the search page. The right column is kept. No portlets will be displayed, it is used for the facets.

The first thing we do in our search is geting the results for the search query, if there is one:

```
def search(self):
    if not self.request.get('SearchableText'):
        return []
    catalog = api.portal.get_tool('portal_catalog')
    results = IContentListing(catalog(hl='true', **self.request.form))
    self.has_results = bool(len(results))
    b_start = self.request.get('b_start', 0)
    batch = Batch(results, size=20, start=b_start)
    return batch
```

We can use the standard Plone catalo API for getting the results.

Note: Don't use plone.api.content.find because it *fixes* the query to match the indexes defined in Zcatalog and will strip all Solr related query parameters. We don't want that.

After we got the results we wrap it with `IContentListing` to have unified access to them. Finally we create a Batch, to make sure long result sets are batched on our search view.

The next thing we have in our search view is the form itself:

```
<form name="searchform"
      action="search"
      class="searchPage"
      tal:attributes="action request/getURL">
  <input class="searchPage" name="SearchableText" id="acsearch" type="text"
         size="25" list="searchresults" title="Search Site"
         placeholder="Search Site ..."
         tal:attributes="value request/SearchableText|nothing;"/>
  <datalist id="searchresults"/>
  <input class="searchPage searchButton" type="submit" value="Search"/>
  <div tal:define="view nocall: context/@@search-facets | nothing"
       tal:condition="python: view"
       tal:replace="structure view/hiddenfields"/>
</form>
```

We have a input field for used input. For the autocompletion we reference the datalist with the `list` attribute. For the facets we need to render the `hiddenfields` snippet, which is constructed by the `search-facets` view of `collective.solr`. This snippet will add the necessary query parameters like **facet=true&facet.field=portal_type&facet.field=review_state**.

We use the `h1` element for displaying the number of elements.

The next section is reseved for the suggest snippet:

```
<div tal:condition="not: view/has_results">
  <p tal:define="suggest view/suggest">
    <tal:noresuls>No results were found.</tal:noresuls>
    <tal:suggest condition="suggest">Did you mean:
      <strong>
        <a href="" tal:attributes="href suggest/url"
           tal:content="suggest/word">Plone</a>
      </strong>
    </tal:suggest>
  </p>
</div>
```

If no results are found with the query, a term is suggested. This term is fetched from the collective.solr AJAX view **suggest-terms**. The code in our view class is here:

```
def suggest(self):
    self.request.form['term'] = self.request.get('SearchableText')
    suggest_view = getMultiAdapter((self.context, self.request),
                                   name='suggest-terms')
    suggestions = json.loads(suggest_view())
    if suggestions:
        word = suggestions[0]['value']['word']
        query = self.request.form.copy()
        query['SearchableText'] = word
        return {'word': word,
                'url': '{0}?{1}'.format(self.request.getURL(),
                                        urlencode(query, doseq=1))}
    return ''
```

We get suggestions from the Solr handler and construct an URL for a new search with query parameters preserved.

The next thing we have is the result list. There is nothing fancy in it. We show the title, which is linked to the article found and the cropped description.

Finally we have the snippet for the facets in the right slot:

```
<div metal:fill-slot="portlets_two_slot">
    <div tal:define="facet_view nocall: context/@@search-facets;
                       results view/search;"
         tal:condition="view/has_results"
         tal:replace="structure python:facet_view(results=results._sequence._
↪basesequence)"/>
  </div>
```

We call the facet view of `collective.solr` with our resultset and get the facets fully rendered as HTML.

Note: We need to pass the *real* solr response to the facet view. That's why we have to escape the batch (_sequence) and the contentlisting (_basesequence)

Now we have a fully functional Plone search with faceting, autocompletion, suggestion and term highlighting. The complete example you can find on github:

https://github.com/collective/plonetraining.solr_example

Excercise

Have a custom search page with autocomplete, suggest, highlighting and faceting working.

Solr Testing

collective.solr comes with a few test fixtures that make it easier to test Solr.

`SOLR_FIXTURE` fires up and tears down a Solr instance. This fixture can be used to write unit tests for a Solr configuration.

Usually you need the `COLLECTIVE_SOLR_FIXTURE` which spins off a Solr instance and installs `collective.solr`. A custom test layer based on this fixture looks like this:

```
class PlonetrainingSolrExampleLayer(PloneSandboxLayer):

    defaultBases = (COLLECTIVE_SOLR_FIXTURE,)

    def setUpZope(self, app, configurationContext):
        # Load any other ZCML that is required for your tests.
        # The z3c.autoinclude feature is disabled in the Plone fixture base
        # layer.
        self.loadZCML(package=plonetraining.solr_example)

    def setUpPloneSite(self, portal):
        applyProfile(portal, 'plonetraining.solr_example:default')
```

A test for our suggest method in our fancy search looks like this:

```
# -*- coding: utf-8 -*-
"""Setup tests for this package."""
from plone import api
from plone.app.testing import setRoles
from plone.app.testing import TEST_USER_ID
from plonetraining.solr_example.browser.views import FancySearchView
from plonetraining.solr_example.testing import PLONETRAINING_SOLR_EXAMPLE_FUNCTIONAL_TESTING  # noqa
from collective.solr.testing import activateAndReindex
import unittest


class TestSearchView(unittest.TestCase):
    """Test that plonetraining.solr_example is properly installed."""

    layer = PLONETRAINING_SOLR_EXAMPLE_FUNCTIONAL_TESTING

    def setUp(self):
        """Custom shared utility setup for tests."""
        self.portal = self.layer['portal']
        setRoles(self.portal, TEST_USER_ID, ('Manager', ))
        api.content.create(self.portal, 'Document', title='Lorem Ipsum')
        activateAndReindex(self.portal)

    def test_suggest(self):
        """Test if plonetraining.solr_example is installed."""
        request = self.layer['request']
        view = FancySearchView(self.portal, request)
        request.form['SearchableText'] = 'lore'
        self.assertEqual(
            view.suggest(),
            {'url': 'http://nohost?term=lore&SearchableText=lorem', 'word': u'lorem'}
        )
```

Note the **activateAndReindex** method. It is a nice testing helper to cleat the Solr index and reindex all objects again. If testing Solr it is advisable to call it at the test setup. Otherwise the documents created during the tests would pile up in the index.

Exercise

Write a custom test for a Solr feature used in Plone.

Production Setup

Multi Core

Multi core setup is the default for Solr 5 and above but unfortunately not supported by collective.solr. You can access a multicore Solr but only the default core, which can be specified in the `collective.recipe.solrinstance` buildout recipe.

The following options only apply if `collective.recipe.solrinstance:mc` is specified. They are optional if the normal recipe is being used. All options defined in the solr-instance section will we inherited to cores. A core could override a previous defined option.

cores A list of identifiers of Buildout configuration sections that correspond to individual Solr core configurations. Each identifier specified will have the section it relates to processed according to the given options above to generate Solr configuration files for each core.

Each identifier specified will result in a Solr `instanceDir` being created and entries for each core placed in Solr's `solr.xml` configuration.

default-core-name Optional and deprecated. This option controls which core is set as the default for incoming requests that do not specify a core name. This corresponds to the `defaultCoreName` option described at http://wiki.apache.org/solr/CoreAdmin#cores. *No longer* used in Solr 5.

An example for a multi-core configuration you can find in the documentation of `collective.recipe.solrinstance`:

See also:

https://github.com/collective/collective.recipe.solrinstance/blob/master/README.rst#multi-core-solr

Monitoring

collective.solr comes with some predefined munin configurations. The values for munin are collected and exposed via the Java JMX framework.

You will need munin and the jmx_ extension. The procedure is documented here :

See also:

https://github.com/collective/collective.solr/blob/master/docs/usage/monitoring.rst

The munin configs however seem a little outdated.

Different host setup

One use case in a production setup might be the split between the Plone server runs on and the Solr server(s). To make this happen you have to consider a couple of things:

- configure host of Solr in c.solr, it can be done TTW, via ZCML or via /etc/hosts

- make sure the blobstorage directory of Plone is available via a network drive to the Solr host. You need to make sure Solr has read permissions which usually means it has the *SAME* User ID than the user which runs the Zope server.

Further reading

Solr is very well documented in its own wiki.

See also:

https://cwiki.apache.org/confluence/display/solr/Apache+Solr+Reference+Guide

There are a couple of books available.

Alternative Indexing/Search solutions

alm.solrindex

`alm.solrindex` is another addon for connecting Plone search to solr. It takes a different approach:

- `collective.solr` *wraps* the Zope catalog. Each item is indexed both in the ZCatalog and in solr, typically including many indexes in both. When a search is performed, based on the indexes used, it decides to query either ZCatalog or solr but not both.
- `alm.solrindex` operates as an index *within* the Zope catalog, replacing the standard SearchableText index. Solr only needs to index the fulltext, and the ZCatalog no longer needs to do so. When a search is performed that includes a SearchableText criterion, first alm.solrindex will query solr for results, then those results will be further filtered by other ZCatalog indexes.

Pros:

- solr is more efficient than ZCTextIndex at indexing and querying fulltext.
- Avoids duplication of index storage.
- Less data needs to be sent between Plone and solr when indexing.
- Don’t need to add new indexes to solr and reindex.

Cons:

- No admin UI in Plone control panel.
- Customizations can require monkey patching.
- Potential for missing some results. (see below)

Setup

We set up solr in our buildout in a similar way, using the `hexagonit.recipe.download` and `collective.recipe.solr` buildout recipes.

The `solr-instance` buildout part looks a bit different:

```
[solr-instance]
recipe = collective.recipe.solrinstance
solr-location = ${solr-download:location}
host = ${settings:solr-host}
port = ${settings:solr-port}
```

```
basepath = /solr
max-num-results = 500
default-search-field = SearchableText
unique-key = docid
index =
    name:docid          type:integer  stored:true     required:true
    name:SearchableText type:text     stored:false
    name:Title          type:text     stored:false
    name:Description    type:text     stored:false
```

- We set the `unique-key` identifying the record to `docid`. `alm.solrindex` will pass the ZCatalog's internal integer record id (`rid`) in this field.
- We set the `default-search-field` to SearchableText, so that solr queries which don't specify a field will use SearchableText.
- We configure fields for docid and each of the standard Plone fulltext indexes, but not any other fields.
- We set `stored:  false` on the indexes so that solr will only store the docid.

We also need to reference the solr URI in an environment variable for the Plone instance part, so that `alm.solrindex` knows where to connect:

```
[instance]
environment-vars =
SOLR_URI http://${settings:solr-host}:${settings:solr-port}/solr
```

After running buildout, we can start Plone and activate `alm.solrindex` in the Addons control panel.

Note: The default installation profile removes the existing SearchableText, Title, and Description indexes, but does not automatically reindex existing content. If you have existing content in the site, you'll need to do a full reindex of the ZCatalog to get them indexed in solr.

Why are results missing?

There is a limitation to this approach.

solr is configured with a maximum limit on the number of results it will return (`max-num-results` in the buildout configuration). This is done because it hurts performance if there are thousands and thousands of results, and solr has to serialize all of them and Plone has to deserialize all of them.

For queries that only use indexes that are in solr (i.e. the fulltext indexes), this is not a big problem. Solr ranks the results so the limited set it returns should be the most relevant results, and most users are not going to navigate past more than a few pages of results anyway.

However it can be a problem when the search term is very generic (so there are many results and its hard for solr to determine the most relevant ones) and the results are also going to be filtered by other indexes (such as in a faceted search solution). In this case the limited result set from solr is fairly arbitrary, the other filters only get to operate on this limited set, and we might end up missing results that should be there.

Example: Consider a site where there are 10,000 items with the term 'pdf', including one in a folder "/annual-reports/2015". If a search is performed for 'pdf' within the path '/annual-reports/2015':

1. First solr finds all documents matching 'pdf', and ranks them.
2. Next it returns the top 500 results to Plone.

3. Next Plone filters those results by path. There is a good chance that our target document was not included in the 500 that solr returned, so this filters down to no results.

There are a couple workarounds for this problem, both of which have their own tradeoff:

1. Increase `max-num-results` above the total number of documents (but this will hurt performance for queries that return many results).
2. Make sure that other indexes that are likely to narrow down the results a lot are also included in solr (but this detracts from the main advantages of using `alm.solrindex` over `collective.solr`).

Customization

Each type of field has its own *handler* which takes care of translating between ZCatalog and solr queries. These can be overridden to handle advanced customization:

Example: monkey patch the `TextFieldHandler` to use an `edismax` query that allows boosting some fields:

```
from Products.PluginIndexes.common.util import parseIndexRequest
from alm.solrindex.handlers import TextFieldHandler
from alm.solrindex.quotequery import quote_query

def parse_query(self, field, field_query):
    name = field.name
    request = {name: field_query}
    record = parseIndexRequest(request, name, ('query',))
    if not record.keys:
        return None

        query_str = ' '.join(record.keys)
        if not query_str:
            return None

        if name == 'SearchableText':
            q = quote_query(query_str)
        else:
            q = u'+%s:%s' % (name, quote_query(query_str))

        return {
            'q': q,
            'defType': 'edismax',
            'qf': 'Title^10 Description^2 SearchableText^0.2',  # boost fields
            'pf': 'Title~2^20 Description~5^5 SearchableText~10^2',  # boost phrases
        }
        TextFieldHandler.parse_query = parse_query
```

Example: Add a *path* index that works like Zope's `ExtendedPathIndex` (i.e. it'll find anything whose path begins with the query value):

solr.cfg:

```
[solr-instance]
...
index =
    ...
    name:path           type:descendent_path stored:false
```

handlers.py:

```python
from alm.solrindex.handlers import DefaultFieldHandler

class PathFieldHandler(DefaultFieldHandler):

    def parse_query(self, field, field_query):
        query = super(PathFieldHandler, self).parse_query(field, field_query)
        if query == {'fq': 'path:""'}:
            return {}
        return query

    def convert_one(self, value):
        # avoid including the site path in the index data
        if value.startswith('/Plone'):
            value = value[6:]
        return super(PathFieldHandler, self).convert_one(value)
```

ZCML:

```xml
<utility component=".handlers.PathFieldHandler"
         provides="alm.solrindex.interfaces.ISolrFieldHandler"
         name="path" />
```

DIY solr

If both *collective.solr* and *alm.solrindex* are too much for you or you have special needs, you can access Solr by custom code. This might be, if you:

- need to access a Solr server with a newer version / multicore setup and you don't have access to the configuration of Solr
- Only want a fulltext search page of a small site with no need for full realtime support

You can find a full-featured example of a full-fledged custom Solr integration at the Ploneintranet (**advanced!**):

https://github.com/ploneintranet/ploneintranet/pull/299

collective.elasticsearch

Another option for an advanced search integration is the younger project Elasticsearch https://www.elastic.co/products/elasticsearch. Like for Solr, the technical foundation is the Lucene index, written in Java.

Pros of Elasticsearch

- It uses JSON instead of an XML schema for (field) configuration, which might be easier to configure.
- Clustering and replication is builtin from the beginning. It is easier to configure. Especially ad-hoc cluster which can (re)configure automatically.
- The project and community is agile and active.

Cons of Elasticsearch

- JSON is abused as Query DSL. It can lead to queries with up to 10 layers. This can be annoying especially if you write them programatically.

The integration of Elasticsearch with Plone is done with https://pypi.python.org/pypi/collective.elasticsearch/

Google Custom Search

Google provides a couple related tools for using Google as a site-specific search engine embedded in your site: Google Custom Search (free, ad-supported) and Google Site Search (paid).

Note: don't confuse these solutions with Google Search Appliance, which was a rack-mounted device which has been discontinued.

Pros:

- Better ranking of results compared to ZCTextIndex.
- Fairly straightforward to integrate.
- GUI control panel for basic configuration.
- Don't have to run and maintain a separate Java service.
- Can easily be configured to search multiple websites.

Cons:

- Free version includes Google branding and ads in results.
- Cannot index private items.
- Changes are not indexed immediately (usually within a week).
- Only returns top 100 results for a query.
- Only useful for fulltext search, not searching specific fields.
- Limited control over result ranking and formatting.
- Google has a habit of discontinuing free services.

Mastering Plone Workflow

Warning: This chapter is still work in progress!

Controlling security with workflow

Workflow is used in Plone for three distinct, but overlapping purposes:

- To keep track of metadata, chiefly an object's *state*;
- to create content review cycles and model other types of processes;
- to manage object security.

When writing content types, we will often create custom workflows to go with them.

Plone's workflow system is known as DCWorkflow. It is a *states-and-transitions* system, which means that your workflow starts in a particular *state* (the *initial state*) and then moves to other states via *transitions* (also called *actions* in CMF).

When an object enters a particular state (including the initial state), the workflow is given a chance to update **permissions** on the object. A workflow manages a number of permissions – typically the "core" CMF permissions like *View*, *Modify portal content* and so on – and will set those on the object at each state change. Note that this is event-driven, rather than a real-time security check: only by changing the state is the security information updated. This is why you need to click *Update security settings* at the bottom of the `portal_workflow` screen in the ZMI when you change your workflows' security settings and want to update existing objects.

Introduction to Workflows in Plone

What is a Workflow?

Workflow is the series of interactions that should happen to complete a task. Business organizations have many kinds of workflow. For example, insurance companies process claims, delivery companies track shipments, and schools accept applications for admission. All these tasks involve several people, sometimes take a long time, and vary significantly from organization to organization.

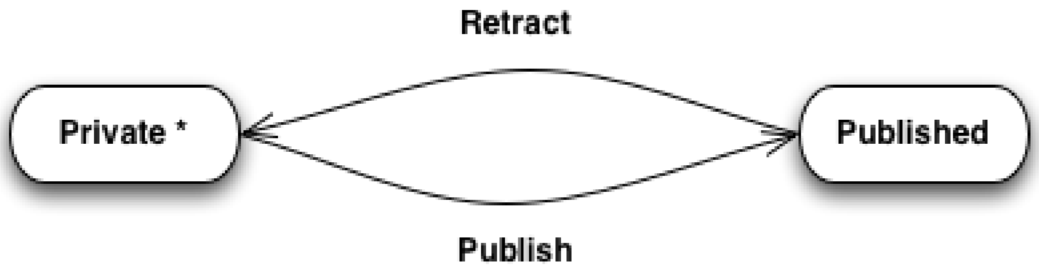
The goal of workflow software is to streamline and track workflow activity. Since different organizations have different workflow processes, workflow software must be flexible and easy to customize.

The workflow system inside of Plone is an example of a State Machine.

From Wikipedia:

> A finite-state machine (FSM) or finite-state automaton (plural: automata), or simply a state machine, is a behavioral model used to design computer programs. It is composed of a finite number of states associated to transitions. A transition is a set of actions that starts from one state and ends in another (or the same) state. A transition is started by a trigger, and a trigger can be an event or a condition.

- Any object controlled by workflow is **always** in precisely one `state` from each workflow in its chain.
- The `state` in which an object is currently located controls what `transitions` are available to it
- **Any workflow can be diagrammed, showing the available states and the transitions between them**
  - Diagrams like this can be of enormous help in understanding your workflow
  - You should always sketch up a diagram when you start figuring out the workflow you want



What's in a Workflow?

Workflows control

- What `states` and `transitions` are available
- Which `permissions` will be managed (permissions not managed are left untouched from their current value by the workflow)
- Which `groups` will be managed (see `states` below for more about this)
- Which `variables` will be tracked by the workflow (values are set and stored every time a transition occurs)

- What `worklists` will be generated (you can return lists of content matching values tracked by `variables`
- What `scripts` are available to be used in conjunction with `transitions`
  - These are basic python scripts, and are not used much anymore now that `events` are available

States control

- What transitions are available **out**
- What `permissions` are assigned to which `roles` *locally* to the object
- What `groups` are assigned to which `roles` *locally* to the object
  - This is probably the least-used aspect of workflow
  - It can be spectacularly useful

Transitions control

- What `state` they will end in
- What conditions or `gaurds` are required for the transition to be available
  - These can be `permissions` of the user, `roles` a user has, `groups` to which the user belongs, or even the boolean value of *'TALES*' expressions
- What `scripts` will be executed before and after the transition occurs (again, not used much now that we have `events`)
- How the transition is triggered
  - This can be user-initiated *or* automatic
    - Automatic transitions happen when an object lands in a state from which they are a valid exit, **and** that object fulfills **all** conditions for the transition to be available.
    - If the conditions for the automated transition are **not** met, then the transition doesn't happen
      - Updating the object to meet the conditions will not kick it off
      - You'll have to back it out of the current `state` and re-do the transition that should have kicked it off

How Does Workflow Work in Plone?

The tool in Plone that handles all workflow is called `portal_workflow`

- Types must be `workflow` aware
  - Types in Plone are made *WorkflowAware* by a base content mixin from CMFCore `WorkflowAware` (in `CMFCatalogAware`
- Workflow is assigned by *type*
  - Each type gets a *chain*
  - A chain can have more than one workflow in it
- `portal_workflow` is responsible for keeping track of all information about the workflow state of an object
  - A particular content object knows **nothing** about it's own workflow state

- queries about the workflow of an object **must** be addressed to portal_workflow

```
>>> from plone import api
>>> fpage = api.content.get("/front-page")
>>> fpage.review_state
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: review_state
>>> api.content.get_state(fpage)
'published'
>>> wft = api.portal.get_tool('portal_workflow')
>>> wft.getChainFor(fpage)
('simple_publication_workflow',)
>>> wft.getTransitionsFor(fpage)
({'description': 'If you submitted the item by mistake or want to perform additional␣
↪edits, this will take it back.', 'title': 'Member retracts submission', 'url':
↪'http://nohost/Plone/front-page/content_status_modify?workflow_action=retract', 'id
↪': 'retract', 'title_or_id': 'Member retracts submission', 'name': 'Retract'}, {
↪'description': 'Sending the item back will return the item to the original author␣
↪instead of publishing it. You should preferably include a reason for why it was not␣
↪published.', 'title': 'Reviewer sends content back for re-drafting', 'url': 'http://
↪nohost/Plone/front-page/content_status_modify?workflow_action=reject', 'id': 'reject
↪', 'title_or_id': 'Reviewer sends content back for re-drafting', 'name': 'Send back
↪'})
>>> with api.env.adopt_user('content'):
...     contrib-page = api.content.create(container=api.portal.get(), type="Document
↪", title="Content Contrib Page")
...     [i['id'] for i in wft.getTransitionsFor(api.content.get("/content-contrib-page
↪")]
...
['submit']
>>> with api.env.adopt_roles(roles=['Manager',]):
...     [i['id'] for i in wft.getTransitionsFor(contrib-page)]
...
['submit', 'publish']
```

- portal_workflow is security conscious, for all aspects of workflow it respects and validates the access levels of the current user
  - Users can only access the workflow information for which they have permissions

```
>>> with api.env.adopt_user('site-admin'):
...     wft.getTransitionsFor(fpage)
...
>>> from pprint import pprint
>>> pprint(wft.getTransitionsFor(fpage))
({'description': 'If you submitted the item by mistake or want to perform
                  additional edits, this will take it back.',
  'id': 'retract',
  'name': 'Retract',
  'title': 'Member retracts submission',
  'title_or_id': 'Member retracts submission',
  'url': 'Plone/front-page/content_status_modify?workflow_action=retract'},
 {'description': 'Sending the item back will return the item to the original
                  author instead of publishing it. You should preferably include
                  a reason for why it was not published.',
  'id': 'reject',
  'name': 'Send back',
  'title': 'Reviewer send content back for re-drafting',
  'title_or_id': 'Reviewer send content back for re-drafting',
  'url': 'Plone/front-page/content_status_modify?workflow_action=reject'})
```

Moving Content Through Workflows

- As stated above, any object with workflow is **always** in exactly **one** `state` for each workflow in it's chain.
  - When you initiate a transition, it is **instantaneous**.
  - What happens when this occurs?
    1. The `BeforeTransitionEvent` is notified, and any subscribers to that event are executed
    2. Any `before script` registered for the transition are executed.
    3. The `transition` takes place
    - values are set for the variables registered by the workflow
    - the new `state` of the object is set
    - the new set of permissions values for roles and groups are calculated and updated
      * first permissions are remapped
      * then group -> role mappings are changed
    - the object is re-indexed for all *security related* indexes.
    4. Any `after script` registered for the transition is executed
    5. The `AfterTransitionEvent` is notified, and any subscribers to that event are executed

  In general, transitions are triggered by user action. This takes place when a user clicks on the *state* menu in the Plone UI and selects an available transition, or when the user presses *save* from the **Change State** dialog found in the folder listing view.

  - As stated above, automatic transitions are found as a result of undergoing manual transitions.
    - Step 3 above can actually be executed **multiple** times when a user triggers a `transition`.
    - Events and scripts are executed for **each** transition that happens
    - For this reason, when subscribing to workflow events, it's a good idea to check *which* transition just happened *before* taking any actions in your handler:

```
def handleWorkflowTransition(ob, event):
    """ a handler meant to be used after a 'publish' transition """
    if event.transition != 'publish':
        return
    ...
```

Basic Roles and Permissions in Plone

Roles, groups, permissions, workflows, states, transitions are all a part of Plone's robust security model. But you don't have to be a guru to understand the basic Plone permissions you will encounter on a daily basis.

Definitions

Let’s start off with some basic terminology. Permissions are individual rights that give the user the ability to perform an action. Roles are a combination of permissions. Both users and groups can be assigned roles.

Roles

Roles are a combination of permissions that you will assign to your users. Plone comes with a basic set of roles, each of which already has certain permissions assigned. Below you will learn a little bit about the defaults for each role.

Most of your site users have the “Member” role. By default, a Member can see anything that is published, see the contents of a folder, see a list of other portal members and groups, and see portlets. Depending on how your site is customized, Members may not have access to certain portlets or specific parts of the site. I keep track of what a Member has access to by reminding myself that a Member cannot change content and can only see what has been published. You will want to assign the Member role to your every day, normal users who will not be changing content. Everyone who joins your site should be assigned this role.

Reader

The Reader role may be almost the opposite from the Member role. Readers can view content items that are in the private state, but cannot make any changes. You should assign people the Reader role when you want them to review a piece of content that is not yet published. The Reader role is great for when you want only certain people to see a piece of content. You can also use the Reader role as part of a document review cycle for users who would like to review your document but not make changes to the document.

Contributor

A user with a Contributor role can do all the things a member can, plus add content, use version control, and view content that is not in the published state. A contributor cannot modify (edit) another user’s content. The Contributor role should be given to users who will create content but not edit another person’s content.

Owner

The owner role is inherited when a user adds a piece of content. You have to have another role, like Contributor, that has the ability to add content. Once you add a piece of content, you are automatically assigned the Owner role over this content. When you are the Owner of a piece of content, you can modify that piece of content whenever you wish, no matter what state the content is in.

Editor

A user with the Editor role by default does not have the ability to add content, but can modify(edit) content and use version control. An Editor can also manage properties of content and can submit content for publication. The Editor role should be used when a Contributor is sending a piece of content for review. The Editor will review, and change, the content and then submit it for publication.

Reviewer

A Reviewer role picks up where the Editor leaves off. While a Reviewer does not have as many rights as the Editor, the Reviewer can publish content that has been sent to the submit for publication state or send it back to the owner. The

Reviewer also has a special portlet just for content that needs to be reviewed. Once an Editor has submitted content for publication, the Reviewer will review the content and then has the option to Publish or send back the content for the Contributor to review. The Reviewer has the final say if something gets published or not.

Site Administrator

The Site Administrator role is very similar to the Manager role described below, but with a few exceptions. The Site Administrator has full access to manage all of the content in the portal, and can perform certain actions from the site setup such as adding and removing users. They do not have access to the ZMI or to actions such as activating Plone add-ons, configuring caching or discussion settings.

Manager

The Manager role is the role that can do everything. A user with the Manager role is a Site Administrator. Manager privileges are not given out lightly as this role can add, delete, and make changes to any thing in the site. While more than one person should have this role, it definitely should not be handed out to large numbers of people. Your site Manger has access to the control panel, where many site wide settings can be changed and updated. The Manager can also manage things via the ZMI (Zope Management Interface).

Giving out permissions

The easiest way to hand out permissions is to assign roles to groups. You can create a group and assign that group a role. Then, whenever you want to give someone certain permissions, you can add that user to that group. Assigning roles on a group level allows you to more easily manage large numbers of users.

Permissions

Plone’s security system is based on the concept of *permissions* protecting *operations* (like accessing a view, viewing a field, modifying a field, or adding a type of content) that are granted to *roles*, which in turn are granted to *users* and/or *groups*. In the context of developing content types, permissions are typically used in three different ways:

- A content type or group of related content types often has a custom *add permission* which controls who can add this type of content.
- Views (including forms) are sometimes protected by custom permissions.
- Individual fields are sometimes protected by permissions, so that some users can view and edit fields that others can’t see.

It is easy to create new permissions. However, be aware that it is considered good practice to use the standard permissions wherever possible and use *workflow* to control which roles are granted these permissions on a per-instance basis.

Standard permissions

Many of the standard permissions can be found in `Product.CMFCore`‘s `permissions.zcml` (`parts/omelette/Products/CMFCore/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 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.

`cmf.AddPortalMember` (*Add portal member*) usually granted to the `Site Administrator` and `Manager` role, controlling the ability to add new users into the site. It is also granted to the `Anonymous` role if you have enabled self user registration.

Here is an example of how Permissions can be changed by event subscribers:

```
>>> from plone import api
>>> api.portal.get_registry_record(name="plone.enable_self_reg")
False
>>> from AccessControl.SecurityManagement import noSecurityManager
>>> noSecurityManager() # Log out the Special System User
>>> api.user.get_current()
<SpecialUser 'Anonymous User'>
>>> api.user.has_permission("Add portal member")
False
>>> api.portal.set_registry_record(name="plone.enable_self_reg", value=True)
>>> api.user.has_permission("Add portal member")
True
```

Inside of *Products.CMFPlone* there is an event subscriber listening for changes to specific registry keys and will alter the permissions in the site based on the change in the setting.

Local Roles

Local Roles on Folders

There may be some situations where you don't want your group to have a specific role across the entire site. You can manage that easily too. When setting up your group in the Site Setup, do not assign it a role. Go to the folder where you want the group to have specific permissions and assign the group that role on the sharing tab for the folder. You can assign individual users permissions at this level as well. Simply add the user to the sharing tab and assign the permission to that user. When you assign roles at an object level like this, you are assigning local roles. Local roles give users (or groups) extra permissions in a very specific context. For example, you may have two groups:

pirates and ninjas. The ninjas probably don't want the pirates mucking about with their content. In this case, you could create a folder for the ninjas and assign their group to have a local role of Owner over the folder. Uncheck the inherit permissions box and now your ninjas have their own folder where they can add content and the pirates cannot see or add anything to this folder. Similarly, if only the pirate captain should have access to a folder, add the pirate captain user to the sharing tab and select the correct permission. Don't forget to uncheck the inherit permissions box, otherwise your folder will inherit permissions from the rest of the site.

Local Roles on Groups

A state can also assign local roles to groups. This is akin to assigning roles to groups on Plone's Sharing tab, but the mapping of roles to groups happens on each state change, much like the mapping of roles to permissions. Thus, you can say that in the pending_secondary state, members of the Secondary reviewers group has the `Reviewer` local role. This is powerful stuff when combined with the more usual role-to-permission mapping.

Additional Resources

- http://docs.plone.org/working-with-content/collaboration-and-workflow/collaboration-through-sharing.html
- http://docs.plone.org/develop/plone/security/local_roles.html

Dynamic Roles

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.

Using Dynamic Roles

To start utilizing dynamic roles in Plone, you will need to create an Zope 3 Adapter for `ILocalRoleProvider` in your custom product that contains the code to return the correct roles for a user in a specific context.

- 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

Note: getRoles() function is called several times per request so you might want to cache the result.

Example `localroles.py`

```
from zope.interface import Interface, implements
from zope.component import adapts
from borg.localrole.interfaces import ILocalRoleProvider

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 getEditorRolesOnContext(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 ["Editor"]

        # 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", ["Editor"]) ]
```

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">

  <adapter
      factory=".localroles.DummyLocalRoleAdapter"
      name="dummy_local_role"
      />

</configure>
```

Placeful Workflow

Sometimes you may want a specific section of the site to allow different permissions and roles than other areas of the site such as providing an intranet are for internal staff to collaborate. In the past, if you wanted to do this, you would need to make custom content types that were identical to the standard types so you can attach an alternate workflow policy to them to limit access. This just causes extra boilerplate code and confusion amongst your users as they just want to create standard “Pages”, but in this area they may have to create “Intranet Pages”.

Plone comes standard with a feature that addresses this specific issue. Plone’s “Workflow Policy Support” add-on is available, but not active by default and allows site administrators to define workflow policies that only apply in specific sections of the site. The name Placeful Workflow comes from the fact you do this in a specific place. Placeful workflow allows you to define workflow policies that define content type to workflow mappings that can be applied in any sub-folder of your Plone site.

Getting Started

To get started with Placeful Workflow in Plone, you will need to first activate the add-on via the `Site Setup` > `Add-Ons` control panel. Just click `Activate` next to the “Workflow Policy Support” add-on and you will be ready to start assigning local policies to folders.

Create or go to any folder inside of your site and click the workflow state menu and you will now see an option for `Policy....` Select this option to begin assigning local workflow mappings to this folder.

By default, the Placeful Workflow product has created some default mappings for you:

- Intranet – Sets the default workflow policy to `Intranet/Extranet`
- Old Plone – Sets the default workflow policy to `Community Workflow`
- One State – Sets the default workflow policy to `Single State Workflow`
- Simple Publication – Sets the default workflow policy to `Simple Publication Workflow`

From the `Workflow Policies` control panel, you can create your own custom mappings and then assign them via the `Policy...` menu option per folder inside your site.

Internals of Placeful Workflow

- Works by providing a more specific `adapter` for the `IWorkflowChain` interface defined by DCWorkflow.
  - This means that when you install this product, the `portal_workflow` tool is marked with an `IPlacefulWorkflow` interface, and from then on, the adapter defined by the product is used when looking up the workflow chain for an object
  - A great example of the marker pattern
- You add a *workflow policy* in the location where you want to have customized workflow assignments.
  - A `policy` is basically just a mapping of workflows to content types. Just like what you see in `ZMI` > `portal_workflow` > `workflows`
  - This policy can control workflow ‘’in” the object where it is located, and *below* it
    - *In* means the policy applies to the object itself and its content.
    - *Below* means that the policy applies only to any contained items (and their contents as well), but not to the original object.
- All this can be handled by GenericSetup as well
  - `portal_placeful_workflow.xml` allows you to declare the presence of policies
  - This is accompanied by a folder of the same name (minus the ‘xml’ part, of course)
    - The folder contains one file per policy: `policy_name.xml` where *policy_name* is replaced by the actual name of your policy

  - Once you've generated a policy, you can add an 'import step' in GenericSetup to use it somewhere (this must be done in code)

portal_placeful_workflow.xml:

```
<?xml version="1.0"?>
<object name="portal_placeful_workflow" meta_type="Placeful Workflow Tool">
 <object name="intranet-content" meta_type="WorkflowPolicy"/>
</object>
```

intranet-content.xml:

```
<?xml version="1.0"?>
<object name="member-content" meta_type="WorkflowPolicy">
 <property name="title">Member Content Policy</property>
 <bindings>
  <default>
   <bound-workflow workflow_id="intranet_workflow"/>
  </default>
  <type default_chain="true" type_id="Document"/>
  <type default_chain="true" type_id="Event"/>
  <type default_chain="true" type_id="Folder"/>
  <type default_chain="true" type_id="Link"/>
  <type default_chain="true" type_id="News Item"/>
  <type default_chain="true" type_id="Topic"/>
 </bindings>
</object>
```

the setup:

```
def set_intranet_workflow_policy(portal):
    # assume code that finds or creates the portal location where the policy should apply
    # the result of this code is 'folder'

    folder.manage_addProduct['CMFPlacefulWorkflow']\
        .manage_addWorkflowPolicyConfig()
    pc = getattr(folder, WorkflowPolicyConfig_id)
    pc.setPolicyIn('intranet-content')
    pc.setPolicyBelow('intranet-content')
```

Multi-chain Workflows

Multiple workflows can be very useful in case you have concurrent processes. For example, an object may be published, but require translation. You can track the review state in the main workflow and the translation state in another. If you index the state variable for the second workflow in the catalog (the state variable is always available on the indexable object wrapper so you only need to add an index with the appropriate name to `portal_catalog`) you can search for all objects pending translation, for example using a *Collection*.

Workflows are mapped to types via the `portal_workflow` tool. There is a default workflow, indicated by the string `(Default)`. Some types have no workflow, which means that they hold no state information and typically inherit permissions from their parent. It is also possible for types to have *multiple workflows*. You can list multiple workflows by separating their names by commas. This is called a *workflow chain*.

Note that in Plone, the workflow chain of an object is looked up by multi-adapting the object and the workflow to the `IWorkflowChain` interface. The adapter factory should return a tuple of string workflow names (`IWorkflowChain` is a specialisation of `IReadSequence`, i.e. a tuple). The default obviously looks at the mappings in the `portal_workflow` tool, but it is possible to override the mapping, e.g. by using a custom adapter registered for some marker interface, which in turn could be provided by a type-specific behavior.

Multiple workflows applied in a single chain co-exist in time. Typically, you need each workflow in the chain to have a different state variable name. The standard `portal_workflow` API (in particular, `doActionFor()`, which is used to change the state of an object) also assumes the transition ids are unique. If you have two workflows in the chain and both currently have a `submit` action available, only the first workflow will be transitioned if you do `portal_workflow.doActionFor(context,'submit')`. Plone will show all available transitions from all workflows in the current object's chain in the `State` drop-down, so you do not need to create any custom UI for this. However, Plone always assumes the state variable is called `review_state` (which is also the variable indexed in `portal_catalog`). Therefore, the state of a secondary workflow won't show up unless you build some custom UI.

In terms of security, remember that the role-to-permission (and group-to-local-role) mappings are event-driven and are set after each transition. If you have two concurrent workflows that manage the same permissions, the settings from the last transition invoked will apply. If they manage different permissions (or there is a partial overlap) then only the permissions managed by the most-recently-invoked workflow will change, leaving the settings for other permissions untouched.

Workflow Variables

Warning: This section is not ready for prime time

State changes result in a number of variables being recorded, such as the actor (the user that invoked the transition), the action (the id of the transition), the date and time and so on. The list of variables is dynamic, so each workflow can define any number of variables linked to TALES expressions that are invoked to calculate the current value at the point of transition. And of course, the workflow keeps track of the current state. The state is exposed as a special type of workflow variable called the state variable. Most workflows in Plone uses the name review_state as the state variable.

Workflow variables are recorded for each state change in the workflow history. This allows you to see when a transition occurred, who effected it, and what state the object was in before or after. In fact, the "current state" of the workflow is internally considered to be the most recent entry in the workflow history.

Workflow variables are also the basis for worklists. They are basically canned queries run against the current state of workflow variables. Plone's review portlet shows all current worklists from all installed workflows. This can be a bit slow, but it does meant that you can use a single portlet to display an amalgamated list of all items on all worklists that apply to the current user. Most Plone workflows have a single worklist that matches on the review_state variable, e.g. showing all items in the pending state.

Using GenericSetup to Manage Plone Workflows

Workflows provide a great amount of flexibility inside of Plone. They have many moving parts such as states, transitions, permissions, variables, worklist and groups. Plone gives you the ability to configure all of these items through the web via the ZMI, but moving these settings to another environment as part of a release or migration can be error prone.

The GenericSetup tool inside of plone, which has the id of `portal_setup`, provides a way to serialize the current state of your workflow polices into a XML file that can be put into your own custom add-on packages. Using this export/import tool, you can now track changes to your workflow polices inside of your source control tools and create releases that allow exact replication of the settings to new or existing environments.

Getting Started

Creating a workflow from scratch using the XML format is tricky at best. It is recommended that you start from the ZMI and copy/paste an existing workflow that most closely matches your business need and use it as a starting point. The changes to the workflow can be configured via the ZMI and then exported to the filesystem for inclusion in your product.

Exporting Workflow Policies

Once your workflow is working locally for your needs, you can export your workflow using the `portal_setup` tool in the ZMI.

1. Login to the ZMI and click `portal_setup`
2. Click the `Export` tab
3. Check the box for `Workflow Tool` (and optionally `Placeful Workflow Policies`)
4. Click the button at the bottom of the page to `Export selected steps`

This will download a files called `setup_tool-[sometimestamp].tar.gz` to your local computer. This tarball will include the `workflows.xml` profile that describes all of the content to workflow policy bindings as well as the export of each workflow policy as an xml file.

These will be the files you will place in your custom product's profiles directory so it can be imported when using it in another instance. You will need to modify the `workflows.xml` file prior to importing since it contains all of the bindings and you will only want to include bindings that are specific to your custom add-on.

Example `workflows.xml`:

```xml
<?xml version="1.0"?>
<object name="portal_workflow">
  <object name="example_workflow" meta_type="Workflow" />
  <object name="example_container_workflow" meta_type="Workflow" />
  <bindings>
    <type type_id="Example Type">
      <bound-workflow workflow_id="example_workflow" />
    </type>
    <type type_id="Example Container">
      <bound-workflow workflow_id="example_container_workflow" />
    </type>
  </bindings>
</object>
```

In this example, the rest of the bindings have been removed so we are only controlling the needed workflows for our product.

The tarball will have a directory called `workflows` that contains each workflow policy for the site. You can remove all of the stock ones and just keep the policies referenced by your `workflows.xml` for import later.

Subsequent updates to your workflow polices can either be made directly on the files system and then re-imported into the site. Or you can make the changes via the ZMI, but you will need to remember to re-export them using this same process and placing the updated files back into your add-on code.

Importing Workflow Policies

There are several options available for re-importing your workflows back into the site. The `portal_setup` tool provides an option for doing a `Tarball Import`, but this doesn't allow you to keep your modified workflows alongside the code in your add-on product. It is recommended that you export your workflow polices using the steps above and place them in your add-on products *default* GenericSetup policy or include them as part of an upgrade step.

Typically, your GenericSetup profiles will be stored in the `profiles` directory of your add-on product. Each subdirectory of the `profiles` directory is usually registered as a separate GenericSetup extension profile or they are used as part of an upgrade step registered to one of these profiles.

Once you have wired the GenericSetup profile folder to your product using ZCML, you can now do the following to import your workflow policies to your current site.

1. Login to the ZMI and click `portal_setup`
2. Click the `Import` tab
3. Select your GenericSetup profile by either *id* or *title*
4. Select how you want to import your profile, if you have run the import already and your policies are part of the profile directly (not upgrade steps), you will want to select the option to `Apply all profiles`
5. Click the button to `Import all steps`

If you only want to run the `Workflow Tool` steps, you will need go to the `Advanced Import Tab` and select your profile by *id* or *title* and then check the box for just `Workflow Tool` and click `Import selected steps`.

Using upgrade profiles is similar, but you will instruct Plone to run a function when upgrading from one version to the next. This function will call up an already registered *migration profile* and run it against the site. These upgrade steps will only run if the version of your product doesn't satisfy the version requirements that were configured via ZCML.

TODO:

- Add a use case story thread that runs through each of the sections to illustrate how each concept works
- Add in more screen shots of the TTW experience of using workflows in plone