Kotti

Release 1.3.0

2018-01-05

Contents

Kotti is a high-level, Pythonic web application framework based on Pyramid and SQLAlchemy. It includes an extensible Content Management System called the Kotti CMS.

If you are a user of a Kotti system, and either found this page through browsing or searching, or were referred here, you will likely want to go directly to the Kotti User Manual.

The documentation below is for developers of Kotti or applications built on top of it.

CHAPTER 1

First Steps

Get an *overview* of what you can do with Kotti, how to *install* it and how to *create* your first Kotti project / add on.

1.1 Overview

Kotti is most useful when you are developing CMS-like applications that

- have complex security requirements,
- use workflows, and/or
- work with hierarchical data.

Built on top of a number of *best-of-breed* software components, most notably Pyramid and SQLAlchemy, Kotti introduces only a few concepts of its own, thus hopefully keeping the learning curve flat for the developer.

1.1.1 Features

You can **try out the default installation** on Kotti's demo page.

The Kotti CMS is a content management system that's heavily inspired by Plone. Its **main features** are:

- **User-friendliness**: editors can edit content where it appears; thus the edit interface is contextual and intuitive
- **WYSIWYG editor**: includes a rich text editor
- **Responsive design**: Kotti builds on Twitter Bootstrap, which looks good both on desktop and mobile
- **Templating**: easily extend the CMS with your own look & feel with little programming required (see *Static resource management*)
- **Add-ons**: install a variety of add-ons and customize them as well as many aspects of the built-in CMS by use of an INI configuration file (see *Configuration*)
- **Security**: the advanced user and permissions management is intuitive and scales to fit the requirements of large organizations

- **Internationalized**: the user interface is fully translatable, Unicode is used everywhere to store data (see *Translations*)

1.1.2 For developers

For developers, Kotti delivers a strong foundation for building different types of web applications that either extend or replace the built-in CMS.

Developers can add and modify through a well-defined API:

- views,
- templates and layout (both via Pyramid),
- *Content types*,
- "portlets" (see `kotti.views.slots`),
- access control and the user database (see *Security*),
- workflows (via repoze.workflow),
- and much more.

Kotti has a **down-to-earth** API. Developers working with Kotti will most of the time make direct use of the Pyramid and SQLAlchemy libraries. Other notable components used but not enforced by Kotti are Colander and Deform for forms, and Chameleon for templating.

Kotti itself is developed on Github. You can check out Kotti's source code via its GitHub repository. Use this command:

```
git clone git@github.com:Kotti/Kotti
```

Continuous testing against different versions of Python and with *PostgreSQL*, *MySQL* and *SQLite* and a complete test coverage make Kotti a **stable** platform to work with. 

1.1.3 Support

- Python 2.7 (Python 3 coming soon)
- Support for PostgreSQL, MySQL and SQLite (tested regularly), and a list of other SQL databases
- Support for WSGI and a variety of web servers, including Apache

1.2 Installation

1.2.1 Requirements

- Python 2.7 (Python 3 will be supported soon)
- virtualenv
- `build_essential` and `python-dev` (on Debian or Ubuntu) or
- `Xcode` (on OS X) or
- equivalent build toolchain for your OS.

1.2.2 Installation using `virtualenv`

It is recommended to install Kotti inside a virtualenv:

```
virtualenv mysite
cd mysite
bin/pip install -r https://raw.github.com/Kotti/Kotti/stable/requirements.txt
bin/pip install Kotti
```

This will install the latest released version of Kotti and all its requirements into your virtualenv.

Kotti uses Paste Deploy for configuration and deployment. An example configuration file is included with Kotti's source distribution. Download it to your virtualenv directory (mysite):

```
wget https://raw.github.com/Kotti/Kotti/stable/app.ini
```

See the list of Kotti tags, perhaps to find the latest released version. You can search the Kotti listing on PyPI also, for the latest Kotti release (Kotti with a capital K is Kotti itself, kotti_this and kotti_that are add-ons in the list on PyPI).

To run Kotti using the `app.ini` example configuration file:

```
bin/pserve app.ini
```

This command runs Waitress, Pyramid's WSGI server, which Kotti uses as a default server. You will see console output giving the local URL to view the site in a browser.

As you learn more, install other servers, with WSGI enabled, as needed. For instance, for Apache, you may install the optional mod_wsgi module, or for Nginx, you may use choose to use uWSGI. See the Pyramid documentation for a variety of server and server configuration options.

The pserve command above uses SQLite as the default database. On first run, Kotti will create a SQLite database called Kotti.db in your mysite directory. Kotti includes support for PostgreSQL, MySQL and SQLite (tested regularly), and other SQL databases. The default use of SQLite makes initial development easy. Although SQLite may prove to be adequate for some deployments, Kotti is flexible for installation of your choice of database during development or at deployment.

1.2.3 Installation using Docker (experimental)

This assumes that you already have Docker installed:

```
docker pull kotti/kotti
docker run -i -t -p 5000:5000 kotti/kotti
```

This should get you a running Kotti instance on port 5000.

1.3 Tutorial

Let's learn by example. In this tutorial, we will:

- create and register a Kotti add-on package
- modify the look and feel of Kotti with a simple CSS example
- add content types
- add forms and custom views

Note: If you have questions going through this tutorial, please post a message to the mailing list or join the #kotti channel on irc.freenode.net to chat with other Kotti users who might be able to help.

The Tutorial assumes you have a virtualenv named `mysite` as described in *Installation*. It is split into three parts:

1.3.1 Tutorial Part 1: Creating an add-on and managing static resources

In the first part of the tutorial, we'll create an add-on package, install and register the package with our site, and use a simple CSS example to learn how Kotti manages static resources.

Kotti add-ons are proper Python packages. A number of them are available on PyPI. They include kotti_media, for adding a set of video and audio content types to a site, kotti_gallery, for adding a photo album content type, kotti_blog, for blog and blog entry content types, etc.

The add-on we will make, `kotti_mysite`, will be just like those, in that it will be a proper Python package created with the same command line tools used to make kotti_media, kotti_blog, and the others. We will set up `kotti_mysite` for our Kotti site, in the same way that we might wish later to install, for example, kotti_media.

So, we are working in the `mysite` directory, a virtualenv, as described in *Installation*. You should be able to start Kotti, and load the front page.

We will create the add-on as `mysite/kotti_mysite`. `kotti_mysite` will be a proper Python package, installable into our virtualenv.

Creating the Add-On Package

To create our add-on, we use the standard Pyramid tool `pcreate`, with `kotti_addon`, a scaffold that was installed as part of Kotti.

```
bin/pcreate -s kotti kotti_mysite
```

The script will ask a number of questions. It is safe to accept the defaults. When finished, observe that a new directory called `kotti_mysite` was added to the current working directory, as `mysite/kotti_mysite`.

Installing Our New Add-On

To install the add-on (or any add-on, as discussed above) into our Kotti site, we'll need to do two things:

- install the package into our virtualenv
- include the package inside our site's `app.ini`

Note: Why two steps? Installation of our add-on as a Python package is different from activating the add-on in our site. Consider that you might have multiple add-ons installed in a virtualenv, but you could elect to activate a subset of them, as you experiment or develop add-ons.

To install the package into the virtualenv, we'll change into the new `kotti_mysite` directory, and issue a `python setup.py develop`. This will install the package in *development mode*:

```
cd kotti_mysite
../bin/python setup.py develop
```

Note: `python setup.py install` is for normal installation of a finished package, but here, for `kotti_mysite`, we will be developing it for some time, so we use `python setup.py develop`. Using this mode, a special link file is created in the site-packages directory of your virtualenv. This link points to the add-on directory, so that any changes you make to the software will be reflected immediately without having to do an install again.

Step two is configuring our Kotti site to include our new `kotti_mysite` package. To do this, open the `app.ini` file, which you downloaded during *Installation*. Find the line that says:

```
kotti.configurators = kotti_tinymce.kotti_configure
```

And add `kotti_mysite.kotti_configure` to it:

```
kotti.configurators =
    kotti_tinymce.kotti_configure
    kotti_mysite.kotti_configure
```

At this point, you should be able to restart the application, but you won't notice anything different. Let's make a simple CSS change and use it to see how Kotti manages static resources.

Static Resources

Kotti uses fanstatic for managing its static resources.

Take a look at `kotti_mysite/kotti_mysite/fanstatic.py` to see how this is done:

```
from fanstatic import Group
from fanstatic import Library
from fanstatic import Resource


library = Library("kotti_mysite", "static")

css = Resource(
    library,
    "styles.css",
    minified="styles.min.css")
js = Resource(
    library,
    "scripts.js",
    minified="scripts.min.js")

css_and_js = Group([css, js])
```

The `css` and `js` resources each define files we can use for our css and js code. We will use `style.css` in our example. Also note the `css_and_js` group. It shows up in the configuration code discussed below.

fanstatic has a number of cool features – you may want to check out their homepage to find out more.

A Simple Example

Let's make a simple CSS change to see how this all works. Open `kotti_mysite/kotti_mysite/static/style.css` and add the following code.

```
h1, h2, h3 {
  text-shadow: 4px 4px 2px #ccc;
}
```

Now, restart the application and reload the front page.

```
cd ..
bin/pserve app.ini
```

Notice how the title has a shadow now?

Configuring the Package with `kotti.configurators`

Remember when we added `kotti_mysite.kotti_configure` to the `kotti.configurators` setting in the `app.ini` configuration file? This is how we told Kotti to call additional code on start-up, so that add-ons have a chance to configure themselves. The function in `kotti_mysite` that is called on application start-up lives in `kotti_mysite/kotti_mysite/__init__.py`. Let's take a look:

```
def kotti_configure(settings):
    ...
    settings['kotti.fanstatic.view_needed'] += ' kotti_mysite.fanstatic.css_and_js'
    ...
```

Here, `settings` is a Python dictionary with all configuration variables in the `[app:kotti]` section of our `app.ini`, plus the defaults. The values of this dictionary are merely strings. Notice how we add to the string `kotti.fanstatic.view_needed`.

Note: Note the initial space in ' kotti_mysite.static.css_and_js'. This allows a handy use of += on different lines. After concatenation of the string parts, blanks will delimit them.

This `kotti.fanstatic.view_needed` setting, in turn, controls which resources are loaded in the public interface (as compared to the edit interface).

As you might have guessed, we could have also completely replaced Kotti's resources for the public interface by overriding the `kotti.fanstatic.view_needed` setting instead of adding to it, like this:

```
def kotti_configure(settings):
    ...
    settings['kotti.fanstatic.view_needed'] = ' kotti_mysite.fanstatic.css_and_js'
    ...
```

This is useful if you've built your own custom theme. Alternatively, you can completely *override the master template* for even more control (e.g. if you don't want to use Bootstrap).

See also *Configuration* for a full list of Kotti's configuration variables, and *Static resource management* for a more complete discussion of how Kotti handles static resources through fanstatic.

In the *next part* of the tutorial, we'll add our first content types, and add forms for them.

1.3.2 Tutorial Part 2: A Content Type

Kotti's default content types include `Document`, `Image` and `File`. In this part of the tutorial, we'll add to these built-in content types by making a `Poll` content type which will allow visitors to view polls and vote on them.

Adding Models

When creating our add-on, the scaffolding added the file `kotti_mysite/kotti_mysite/resources.py`. If you open `resources.py` you'll see that it already contains code for a sample content type `CustomContent` along with the following imports that we will use.

```
from kotti.resources import Content
from sqlalchemy import Column
from sqlalchemy import ForeignKey
from sqlalchemy import Integer
```

Add the following definition for the `Poll` content type to `resources.py`.

```
class Poll(Content):
    id = Column(Integer(), ForeignKey('contents.id'), primary_key=True)

    type_info = Content.type_info.copy(
        name=u'Poll',
        title=u'Poll',
        add_view=u'add_poll',
        addable_to=[u'Document'],
    )
```

Things to note here:

- Kotti's content types use SQLAlchemy for definition of persistence.
- `Poll` derives from *`kotti.resources.Content`*, which is the common base class for all content types.
- `Poll` declares a `sqlalchemy.Column` `id`, which is required to hook it up with SQLAlchemy's inheritance.
- The `type_info` class attribute does essential configuration. We refer to name and title, two properties already defined as part of `Content`, our base class. The `add_view` defines the name of the add view, which we'll come to in a second. Finally, `addable_to` defines which content types we can add `Poll` items to.
- We do not need to define any additional `sqlalchemy.Column` properties, as the `title` is the only property we need for this content type.

We'll add another content class to hold the choices for the poll. Add this into the same `resources.py` file:

```
class Choice(Content):
    id = Column(Integer(), ForeignKey('contents.id'), primary_key=True)
    votes = Column(Integer())

    type_info = Content.type_info.copy(
        name=u'Choice',
        title=u'Choice',
        add_view=u'add_choice',
        addable_to=[u'Poll'],
    )

    def __init__(self, votes=0, **kwargs):
        super(Choice, self).__init__(**kwargs)
        self.votes = votes
```

The `Choice` class looks very similar to `Poll`. Notable differences are:

- It has an additional sqla.Column property called `votes`. We'll use this to store how many votes were given for the particular choice. We'll again use the inherited `title` column to store the title of our choice.

- The `type_info` defines the title, the `add_view` view, and that choices may only be added *into* `Poll` items, with the line `addable_to=[u'Poll']`.

Adding Forms and a View

Views (including forms) are typically put into a module called `views`. The Kotti scaffolding further separates this into `view` and `edit` files inside a `views` directory.

Open the file at `kotti_mysite/kotti_mysite/views/edit.py`. It already contains code for the `CustomContent` sample content type. We will take advantage of the imports already there.

```
import colander
from kotti.views.edit import ContentSchema
from kotti.views.form import AddFormView
from kotti.views.form import EditFormView
from pyramid.view import view_config

from kotti_mysite import _
```

Some things to note:

- Colander is the library that we use to define our schemas. Colander allows us to validate schemas against form data.
- Our class inherits from `kotti.views.edit.ContentSchema` which itself inherits from `colander.MappingSchema`.
- `_` is how we hook into i18n for translations.

Add the following code to `views/edit.py`:

```
class PollSchema(ContentSchema):
    """Schema for Poll"""

    title = colander.SchemaNode(
        colander.String(),
        title=_(u'Question'),
    )


class ChoiceSchema(ContentSchema):
    """Schema for Choice"""

    title = colander.SchemaNode(
        colander.String(),
        title=_(u'Choice'),
    )
```

The two classes define the schemas for our forms. The schemas specify which fields we want to display in the forms. We want to display the `title` field.

Let's move on to building the actual forms. Add this to `views/edit.py`:

```
from kotti_mysite.resources import Choice
from kotti_mysite.resources import Poll


@view_config(name='edit', context=Poll, permission='edit',
             renderer='kotti:templates/edit/node.pt')
```

```
class PollEditForm(EditFormView):
    schema_factory = PollSchema


@view_config(name=Poll.type_info.add_view, permission='add',
             renderer='kotti:templates/edit/node.pt')
class PollAddForm(AddFormView):
    schema_factory = PollSchema
    add = Poll
    item_type = u"Poll"


@view_config(name='edit', context=Choice, permission='edit',
             renderer='kotti:templates/edit/node.pt')
class ChoiceEditForm(EditFormView):
    schema_factory = ChoiceSchema


@view_config(name=Choice.type_info.add_view, permission='add',
             renderer='kotti:templates/edit/node.pt')
class ChoiceAddForm(AddFormView):
    schema_factory = ChoiceSchema
    add = Choice
    item_type = u"Choice"
```

Using the `AddFormView` and `EditFormView` base classes from Kotti, these forms are simple to define. We associate the schemas defined above, setting them as the `schema_factory` for each form, and we specify the content types to be added by each.

We use `@view_config` to add our views to the application. This takes advantage of a `config.scan()` call in `__init__.py` discussed below. Notice that we can declare `permission`, `context`, and a `template` for each form, along with its `name`.

Wiring up the Content Types and Forms

Before we can see things in action, we need to add a reference to our new content types in `kotti_mysite/kotti_mysite/__init__.py`.

Open `__init__.py` and modify the `kotti_configure` method so that the `settings['kotti.available_types']` line looks like this.

```
  def kotti_configure(settings):
        ...
      settings['pyramid.includes'] += ' kotti_mysite'
      settings['kotti.available_types'] += (
          ' kotti_mysite.resources.Poll' +
          ' kotti_mysite.resources.Choice')
      settings['kotti.fanstatic.view_needed'] += (
          ' kotti_mysite.fanstatic.css_and_js')
      ...
```

Here, we've added our two content types to the site's `available_types`, a global registry. We also removed the `CustomContent` content type included with the scaffolding.

Notice the `includeme` method at the bottom of `__init__.py`. It includes the call to `config.scan()` that we mentioned above while discussing the `@view_config` statements in our views.

```
def includeme(config):
    ...
    config.scan(__name__)
```

You can see the Pyramid documentation for scan for more information.

Adding a Poll and Choices to the site

Let's try adding a Poll and some choices to the site. Start the site up with the command

```
bin/pserve app.ini
```

Login with the username *admin* and password *qwerty* and click on the Add menu button. You should see a few choices, namely the base Kotti classes `Document`, `File` and `Image` and the Content Type we added, `Poll`.

Lets go ahead and click on `Poll`. For the question, let's write *"What is your favourite color?"*. Now let's add three choices, *"Red"*, *"Green"* and *"Blue"* in the same way we added the poll. Remember that you must be in the context of the poll to add each choice.

If we now go to the poll we added, we can see the question, but not our choices, which is definitely not what we wanted. Let us fix this, shall we?

Adding a custom View to the Poll

First, we need to write a view that will send the needed data (in our case, the choices we added to our poll). Here is the code, added to `view.py`.

```
from kotti_mysite.fanstatic import css_and_js
from kotti_mysite.resources import Poll


@view_defaults(context=Poll)
class PollViews(BaseView):
    """ Views for :class:`kotti_mysite.resources.Poll` """

    @view_config(name='view', permission='view',
                 renderer='kotti_mysite:templates/poll.pt')
    def poll_view(self):
        css_and_js.need()
        choices = self.context.children
        return {
            'choices': choices,
        }
```

Since we want to show all `Choices` added to a `Poll` we can simply use the `children` attribute. This will return a list of all the 'children' of a `Poll` which are exactly the `Choices` added to that particular `Poll`. The view returns a dictionary of all choices under the keyword *'choices'*. The keywords a view returns are automatically available in it's template.

Next on, we need a template to actually show our data. It could look something like this. Create a folder named `templates` and put the file `poll.pt` into it.

```
<!DOCTYPE html>
<html xmlns:tal="http://xml.zope.org/namespaces/tal"
      xmlns:metal="http://xml.zope.org/namespaces/metal"
      metal:use-macro="api.macro('kotti:templates/view/master.pt')">
```

```
  <article metal:fill-slot="content" class="poll-view content">
    <h1>${context.title}</h1>
    <ul>
        <li tal:repeat="choice choices">${choice.title}</li>
    </ul>
  </article>

</html>
```

The first 6 lines are needed so our template plays nicely with the master template (so we keep the add/edit bar, base site structure etc.). The next line prints out the context.title (our question) inside the `<h1>` tag and then prints all choices (with links to the choice) as an unordered list.

Note: We are using two 'magically available' attributes in the template - `context` and `choices`.

- `context` is automatically available in all templates and as the name implies it is the context of the view (in this case the `Poll` we are currently viewing).
- `choices` is available because we sent it to the template in the Python part of the view. You can of course send multiple variables to the template, you just need to return them in your Python code.

With this, we are done with the second tutorial. Restart the application, take a look at the new `Poll` view and play around with the template until you are completely satisfied with how our data is presented.

Note: If you will work with templates for a while (or any time you're developing basically) using the pyramid *'reload_templates'* and *'debug_templates'* options is recommended, as they allow you to see changes to the template without having to restart the application. These options need to be put in your configuration INI under the *'[app:kotti]'* section.

```
[app:kotti]
pyramid.reload_templates = true
pyramid.debug_templates = true
```

In the *next tutorial*, we will learn how to enable our users to actually vote for one of the `Poll` options.

1.3.3 Tutorial Part 3: User interaction

In this part of the tutorial, we will change the site we made in the previous one so our users can actually vote on our polls.

Enabling voting on Poll Choices

We will enable users to vote using a new view. When the user goes to that link, his or her vote will be saved and they will be redirected back to the Poll.

First, let's construct a new view. As before, add the following code to `kotti_mysite/kotti_mysite/views/view.py`.

```
from kotti_mysite.resources import Choice
from pyramid.httpexceptions import HTTPFound
```

```
@view_defaults(context=Choice)
class ChoiceViews(BaseView):
    """ Views for :class:`kotti_mysite.resources.Choice` """

    @view_config(name='vote', permission='view')
    def vote_view(self):
        self.context.votes += 1
        return HTTPFound(
            location=self.request.resource_url(self.context.parent))
```

The view will be called on the `Choice` content type, so the context is the `Choice` itself. We add 1 to the current votes of the `Choice`, then we do a redirect using `pyramid.httpexceptions.HTTPFound`. The location is the parent of our context - the `Poll` in which our `Choice` resides.

With this, we can now vote on a `Choice` by appending `/vote` at the end of the `Choice` URL.

Changing the Poll view so we see the votes

First, we will add some extra content into our `poll_view` so we are able to show the distribution of votes across all choices.

```
def poll_view(self):
    css_and_js.need()
    choices = self.context.values()
    all_votes = sum(choice.votes for choice in choices)
    return {
        'choices': choices,
        'all_votes': all_votes
    }
```

Our view will now be able to get the sum of all votes in the poll via the `all_votes` variable. We will also want to change the choices list to link to our new vote view. Open `poll.pt` and change the link into:

```
...
<li tal:repeat="choice choices">
  <a href="${request.resource_url(choice)}vote">
    ${choice.title}
  </a> (${choice.votes}/${all_votes})
</li>
...
```

This will add the number of votes/all_votes after each choice and enable us to vote by clicking on the choice. Fire up the server and go test it now.

Adding an info block about voting on the view

As you can see, the voting now works, but it doesn't look particularly good. Let us at least add a nice information bubble when we vote. The easiest way to go about that is to use `request.session.flash`, which allows us to flash different messages (success, error, info etc.). Change the `vote_view` to include the the flash message before redirecting.

```
def vote_view(self):
    self.context.votes += 1
    self.request.session.flash(
```

```
        u'You have just voted for the choice "{0}"'.format(
            self.context.title), 'info')
    return HTTPFound(
        location=self.request.resource_url(self.context.parent))
```

Note: Don't forget that since we changed the Python code, we need to restart the application, even if we enabled template reloading and debugging!

As before, you are encouraged to play around a bit more, as you learn much by trying out new things. A few ideas on what you could work on are:

- Change the Choice content type so it has an extra description field that is not required (if you change database content, you will need to delete the database or do a migration). Then make a new Choice view that will list the extra information.
- Make sure only authenticated users can vote, anonymous users should see the results but when trying to vote, it should move them to the login page. Also make sure that each user can vote only once, and list all users who voted for the Choice on the Choice's view.

CHAPTER 2

Narrative Documentation

The narrative documentation contains various topics that explain how to use Kotti.

2.1 Basic Topics

2.1.1 Developer manual

Read the *Configuration* section first to understand which hooks both integrators and developers can use to customize and extend Kotti.

Contents

Screencast tutorial

Here's a screencast that guides you through the process of creating a simple Kotti add-on for visitor comments:

Content types

Defining your own content types is easy. The implementation of the Document content type serves as an example here:

```
from kotti.resources import Content

class Document(Content):
    id = Column(Integer(), ForeignKey('contents.id'), primary_key=True)
    body = Column(UnicodeText())
    mime_type = Column(String(30))

    type_info = Content.type_info.copy(
        name=u'Document',
        title=_(u'Document'),
        add_view=u'add_document',
        addable_to=[u'Document'],
        )

    def __init__(self, body=u"", mime_type='text/html', **kwargs):
        super(Document, self).__init__(**kwargs)
        self.body = body
        self.mime_type = mime_type
```

The `add_view` parameter of the `type_info` attribute is the name of a view that can be used to construct a `Document` instance. This view has to be available for all content types specified in `addable_to` parameter. See the section below and the *Adding Forms and a View* section in the tutorial on how to define a view restricted to a specific context.

You can configure the list of active content types in Kotti by modifying the *kotti.available_types* setting.

Note that when adding a relationship from your content type to another Node, you will need to add a `primaryjoin` parameter to your relationship. An example:

```
from sqlalchemy.orm import relationship

class DocumentWithRelation(Document):
  id = Column(Integer, ForeignKey('documents.id'), primary_key=True)
  related_item_id = Column(Integer, ForeignKey('nodes.id'))
  related_item = relationship(
      'Node', primaryjoin='Node.id==DocumentWithRelation.related_item_id')
```

Add views, subscribers and more

pyramid.includes allows you to hook `includeme` functions that you can use to add views, subscribers, and more aspects of Kotti. An `includeme` function takes the *Pyramid Configurator API* object as its sole argument.

Here's an example that'll override the default view for Files:

```
def my_file_view(request):
    return {...}

def includeme(config):
    config.add_view(
        my_file_view,
        name='view',
        permission='view',
```

```
        context=File,
        )
```

To find out more about views and view registrations, please refer to the Pyramid documentation.

By adding the *dotted name string* of your `includeme` function to the *pyramid.includes* setting, you ask Kotti to call it on application start-up. An example:

```
pyramid.includes = mypackage.views.includeme
```

Working with content objects

Every content node in the database (be it a document, a file...) is also a container for other nodes. You can access, add and delete child nodes of a node through a dict-like interface. A node's parent may be accessed through the `node.__parent__` property.

`kotti.resources.get_root` gives us the root node:

```
>>> from kotti.resources import get_root
>>> root = get_root()
>>> root.__parent__ is None
True
>>> root.title = u'A new title'
```

Let us add three documents to our root:

```
>>> from kotti.resources import Document
>>> root['first'] = Document(title=u'First page')
>>> root['second'] = Document(title=u'Second page')
>>> root['third'] = Document(title=u'Third page')
```

Note how the keys in the dict correspond to the `name` of child nodes:

```
>>> first = root['first']
>>> first.name
u'first'
>>> first.__parent__ == root
True
>>> third = root['third']
```

We can make a copy of a node by using the `node.copy()` method. We can delete child nodes from the database using the `del` operator:

```
>>> first['copy-of-second'] = root['second'].copy()
>>> del root['second']
```

The lists of keys and values are ordered:

```
>>> root.keys()
[u'first', u'third']
>>> first.keys()
[u'copy-of-second']
>>> root.values()
[<Document ... at /first>, <Document ... at /third>]
```

There's the `node.children` attribute should you ever need to change the order of the child nodes. `node.children` is a SQLAlchmey `ordered_list` which keeps track of the order of child nodes for us:

```
>>> root.children
[<Document ... at /first>, <Document ... at /third>]
>>> root.children[:] = [root.values()[-1], root.values()[0]]
>>> root.values()
[<Document ... at /third>, <Document ... at /first>]
```

Note: Removing an element from the `nodes.children` list will not delete the child node from the database. Use `del node[child_name]` as above for that.

You can move a node by setting its `__parent__`:

```
>>> third.__parent__
<Document ... at />
>>> third.__parent__ = first
>>> root.keys()
[u'first']
>>> first.keys()
[u'copy-of-second', u'third']
```

Also see:

- *kotti.views.slots*
- *kotti.events*

kotti.configurators

Requiring users of your package to set all the configuration settings by hand in the Paste Deploy INI file is not ideal. That's why Kotti includes a configuration variable through which extending packages can set all other INI settings through Python. Here's an example of a function that programmatically modified `kotti.base_includes` and `kotti.principals_factory` which would otherwise be configured by hand in the INI file:

```
# in mypackage/__init__.py
def kotti_configure(config):
    config['kotti.base_includes'] += ' mypackage.views'
    config['kotti.principals_factory'] = 'mypackage.security.principals'
```

And this is how your users would hook it up in their INI file:

```
kotti.configurators = mypackage.kotti_configure
```

Security

Kotti uses Pyramid's security API, most notably its support inherited access control lists support. On top of that, Kotti defines *roles* and *groups* support: Users may be collected in groups, and groups may be given roles, which in turn define permissions.

The site root's ACL defines the default mapping of roles to their permissions:

```
root.__acl__ == [
    ['Allow', 'system.Everyone', ['view']],
    ['Allow', 'role:viewer', ['view']],
    ['Allow', 'role:editor', ['view', 'add', 'edit']],
```

```
    ['Allow', 'role:owner', ['view', 'add', 'edit', 'manage']],
    ]
```

Every Node object has an `__acl__` attribute, allowing the definition of localized row-level security.

The *kotti.security.set_groups()* function allows assigning roles and groups to users in a given context. *kotti.security.list_groups()* allows one to list the groups of a given user. You may also set the list of groups globally on principal objects, which are of type *kotti.security.Principal*.

Kotti delegates adding, deleting and search of user objects to an interface it calls *kotti.security.AbstractPrincipals*. You can configure Kotti to use a different `Principals` implementation by pointing the `kotti.principals_factory` configuration setting to a different factory. The default setting here is:

```
kotti.principals_factory = kotti.security.principals_factory
```

There are views that you might want to override when you override the principal factory. That is, if you use different columns in the database, then you will probably want to make changes to the deform schema as well.

These views are `kotti.views.users.UsersManage`, `kotti.views.users.UserManage` and `kotti.views.users.Preferences`. Notice that you should override them using the standard way, that is, by overriding `setup-users`, `setup-user` or `prefs` views. Then you can override any sub-view used inside them as well as include any logic for your usecase when it is called, if needed.

2.1.2 Security

Kotti security is based on the concepts of users, groups, roles, permissions and workflow.

User A user is an entity that can authenticate himself.

Group A group is a collection of users or other groups.

Permission A permission describes what is allowed on an object.

Permissions are never directly assigned to users or groups but always aggregated in roles.

Role A Role is a collection of permissions.

Users or groups can have global or local roles.

Global Roles Global roles are assigned to a user or group via Kotti's user management screens. They apply to every object in a site. You should use them very rarely, maybe only assign the "Adminsitrator" role to the "Administrator" group. This assignment is present by default in a fresh Kotti site.

Local Roles Local roles are assigned to a user or group via the "Sharing" screen of a content object. They apply only to this object and its children.

Workflow The workflow keeps track of the current state of each object lifecycle to manage content security. There is an initial state and you can move to other states thanks to transitions; each state defines a security matrix with roles and permissions. By default Kotti provides a two-state workflow (private and public) for all object types except files and images. Kotti's workflow implementation is based on repoze.workflow.

How to create a new role

Small recipe you can use if you want to create a new role:

```
from kotti.security import (
    Principal,
    ROLES,
    SHARING_ROLES,
```

```
    set_roles,
    set_sharing_roles,
    set_user_management_roles,
    )
from kotti_yourpackage import _


def add_role(role_id, role_title):
    """ Add role in share view and user management views """
    UPDATED_ROLES = ROLES.copy()
    UPDATED_ROLES[role_id] = Principal(role_id,
                                       title=role_title)
    UPDATED_SHARING_ROLES = list(SHARING_ROLES)
    UPDATED_SHARING_ROLES.append(role_id)
    set_roles(UPDATED_ROLES)
    set_sharing_roles(UPDATED_SHARING_ROLES)
    set_user_management_roles(UPDATED_SHARING_ROLES + ['role:admin'])


add_role(u'role:customer', _(u'Customer'))
```

Practically you can add the code above to any file, as long as it is imported on application startup. However, good practice would be to add it to your add on's `__init__.py` for small amounts of changes (like in the example) or to a separate file for larger amounts.

Workflows

You can use an XML file (zcml) in order to describe your workflow. You can see an example here: workflow.zcml.

As you can see it is quite straightforward to add new states, transitions, permissions, etc. You can easily turn the default 2-state website workflow into something completely different or turn your Kotti app into an intranet application.

The default workflow definition is loaded from your project's `.ini` file (using the `kotti.use_workflow` setting). The `kotti.use_workflow` setting's default value is:

```
kotti.use_workflow = kotti:workflow.zcml
```

You can change change the default workflow for your site, register new workflows related to specific content types or disable it completely.

How to disable the default workflow

Kotti is shipped with a simple workflow definition based on private and public states. If your particular use case does not require workflows at all, you can disable this feature with a non true value. For example:

```
kotti.use_workflow = 0
```

How to override the default workflow for all content types

The default workflow is quite useful for websites, but sometimes you need something different. Just point the `kotti.use_workflow` setting to your zcml file:

```
kotti.use_workflow = kotti_yourplugin:workflow.zcml
```

The simplest way to deal with workflow definitions is:

1. create a copy of the default workflow definition and
2. customize it (change permissions, add new states, permissions, transitions, initial state and so on).

If you change workflow settings, you need to reset all your content's workflow states and thus the permissions for all objects under workflow control using the `kotti-reset-workflow` console script.

kotti-reset-workflow command usage

If you change workflow settings you'll need to update security.

```
$ kotti-reset-workflow --help
Reset the workflow of all content objects in the database.

    This is useful when you want to migrate an existing database to
    use a different workflow.  When run, this script will reset all
    your content objects to use the new workflow, while trying to
    preserve workflow state information.

    For this command to work, all currently persisted states must map
    directly to a state in the new workflow.  As an example, if
    there's a 'public' object in the database, the new workflow must
    define 'public' also.

    If this is not the case, you may choose to reset all your content
    objects to the new workflow's *initial state* by passing the
    '--purge-existing' option.

    Usage:
      kotti-reset-workflow <config_uri> [--purge-existing]

    Options:
      -h --help          Show this screen.
      --purge-existing   Reset all objects to new workflow's initial state.
```

How to enable the standard workflow for images and files

Images and files are not associated with the default workflow. If you need a workflow for these items you need to attach the `IDefaultWorkflow` marker interface.

You can add the following lines in your includeme function:

```
from zope.interface import implementer
from kotti.interfaces import IDefaultWorkflow
from kotti.resources import File
from kotti.resources import Image
...

def includeme(config):
    ...
    # enable workflow for images and files
    implementer(IDefaultWorkflow)(Image)
    implementer(IDefaultWorkflow)(File)
    ...
```

How to assign a different workflow to a content type

We are going to use the default workflow for standard content types and a custom workflow for content types providing the `ICustomContent` marker interface. All other content types will still use the default workflow. Third party developers will be able to override our custom workflow without having to touch any line of code (just a `.ini` configuration file)

Let's assume you are starting with a standard Kotti package created with `pcreate -s kotti kotti_wf`.

Four steps are needed:

1. create a new marker interface ICustomContent,
2. change `kotti_wf.resource` (replace `IDefaultWorkflow` with our new `ICustomContent`),
3. create the new workflow definition and
4. register your workflow definition.

Create a new module `kotti_wf/interfaces.py` with this code. This is **optional** but it doesn't hurt, the important thing is to omit the `IDefaultWorkflow` implementer from `kotti_wf.resources`:

```
from zope.interface import Interface


class ICustomContent(Interface):
    """ Custom content marker interface """
```

Change your `kotti_wf.resources` module like so:

```
from kotti.resources import Content
from zope.interface import implements

from kotti_wf.interfaces import ICustomContent


class CustomContent(Content):
    """ A custom content type. """

    implements(ICustomContent)
```

Here it is, our "custom" workflow definition assigned to our `ICustomContent` marker interface:

```
<configure xmlns="http://namespaces.repoze.org/bfg"
           xmlns:i18n="http://xml.zope.org/namespaces/i18n"
           i18n:domain="Kotti">

  <include package="repoze.workflow" file="meta.zcml"/>

  <workflow
      type="security"
      name="custom"
      state_attr="state"
      initial_state="private"
      content_types="kotti_wf.interfaces.ICustomContent"
      permission_checker="pyramid.security.has_permission"
      >

    <state name="private" callback="kotti.workflow.workflow_callback">
```

```
      <key name="title" value="_(u'Private')" />
      <key name="order" value="1" />

      <key name="inherit" value="0" />
      <key name="system.Everyone" value="" />
      <key name="role:viewer" value="view" />
      <key name="role:editor" value="view add edit delete state_change" />
      <key name="role:owner" value="view add edit delete manage state_change" />

    </state>

  </workflow>

</configure>
```

Last you have to tell Kotti to register your new custom workflow including our `zcml` file:

```
kotti.zcml_includes = kotti_wf:workflow.zcml
```

Special cases:

- if you change workflow settings on a site with existing `CustomContent` instances, you need to update the workflow settings using the `kotti-reset-workflow` command.
- if you assign a new workflow definition to a content that already provides the `IDefaultWorkflow` marker interface (by default all content types except files and images), you will have to create and attach on your workflow definition an `elector` function (it is just a function accepting a context and returning `True` or `False`)

2.1.3 Configuration

Contents

- *URL normalization*
- *Local navigation*

INI File

Kotti is configured using an INI configuration file. The *Installation* section explains how to get hold of a sample configuration file. The `[app:kotti]` section in it might look like this:

```
[app:kotti]
use = egg:Kotti
pyramid.reload_templates = true
pyramid.debug_authorization = false
pyramid.debug_notfound = false
pyramid.debug_routematch = false
pyramid.debug_templates = true
pyramid.default_locale_name = en
pyramid.includes = pyramid_debugtoolbar
                   pyramid_tm
mail.default_sender = yourname@yourhost
sqlalchemy.url = sqlite:///%(here)s/Kotti.db
kotti.site_title = Kotti
kotti.secret = changethis1
```

Various aspects of your site can be changed right here.

Overview of settings

This table provides an overview of available settings. All these settings must go into the `[app:kotti]` section of your Paste Deploy configuration file.

Only the settings in bold letters required. The rest has defaults.

Do take a look at the required settings (in bold) and adjust them in your site's configuration. A few of the settings are less important, and sometimes only used by developers, not integrators.

Setting	Description
kotti.site_title	The title of your site
kotti.secret	Secret token used for the initial admin password
kotti.secret2	Secret token used for email password reset token
sqlalchemy.url	SQLAlchemy database URL
mail.default_sender	Sender address for outgoing email
kotti.asset_overrides	Override Kotti's templates
kotti.authn_policy_factory	Component used for authentication
kotti.authz_policy_factory	Component used for authorization
kotti.available_types	List of active content types
kotti.base_includes	List of base Python configuration hooks
kotti.caching_policy_chooser	Component for choosing the cache header policy
kotti.configurators	List of advanced functions for config
kotti.date_format	Date format to use, default: `medium`
kotti.datetime_format	Datetime format to use, default: `medium`
kotti.depot_mountpoint	Configure the mountpoint for the blob storage. See *Working with Blob Data in Kotti* for deta

Con

Table 2.1 – continued from previous page

Setting	Description
kotti.depot_replace_wsgi_file_wrapper	Replace you WSGI server's file wrapper with `pyramid.response.FileIter`.
kotti.depot.*.*	Configure the blob storage. See *Working with Blob Data in Kotti* for details.
kotti.fanstatic.edit_needed	List of static resources used for edit interface
kotti.fanstatic.view_needed	List of static resources used for public interface
kotti.login_success_callback	Override Kotti's default `login_success_callback` function
kotti.max_file_size	Max size for file uploads, default: `10` (MB)
kotti.modification_date_excludes	List of attributes in dotted name notation that should not trigger an update of `modificati`
kotti.populators	List of functions to fill initial database
kotti.request_factory	Override Kotti's default request factory
kotti.reset_password_callback	Override Kotti's default `reset_password_callback` function
kotti.root_factory	Override Kotti's default Pyramid *root factory*
kotti.sanitize_on_write	Configure *Sanitizers* to be used on write access to resource objects
kotti.sanitizers	Configure available *Sanitizers*
kotti.search_content	Override Kotti's default search function
kotti.session_factory	Component used for sessions
kotti.templates.api	Override `api` object available in templates
kotti.time_format	Time format to use, default: `medium`
kotti.url_normalizer	Component used for url normalization
kotti.zcml_includes	List of packages to include the ZCML from
mail.host	Email host to send from
pyramid.default_locale_name	Set the user interface language, default `en`
pyramid.includes	List of Python configuration hooks

kotti.secret and kotti.secret2

The value of `kotti.secret` will define the initial password of the `admin` user. Thus, if you define `kotti.secret = mysecret`, the admin password will be `mysecret`. Log in and change the password at any time through the web interface.

The `kotti.secret` token is also used for signing browser session cookies. The `kotti.secret2` token is used for signing the password reset token.

Here's an example:

```
kotti.secret = myadminspassword
kotti.secret2 = $2a$12$VVpW/i1MA2wUUIUHwY6v8O
```

Note: Do not use these values in your site

Override templates (`kotti.asset_overrides`)

In your settings file, set `kotti.asset_overrides` to a list of *asset specifications*. This allows you to set up a directory in your package that will mirror Kotti's own and that allows you to override Kotti's templates on a case by case basis.

As an example, image that we wanted to override Kotti's master layout template. Inside the Kotti source, the layout template is located at `kotti/templates/view/master.pt`. To override this, we would add a directory to our own package called `kotti-overrides` and therein put our own version of the template so that the full path to our own custom template is `mypackage/kotti-overrides/templates/view/master.pt`.

We can then register our `kotti-overrides` directory by use of the `kotti.asset_overrides` setting, like so:

```
kotti.asset_overrides = mypackage:kotti-overrides/
```

Use add-ons

Add-ons will usually include in their installation instructions which settings one should modify to activate them. Configuration settings that are used to activate add-ons are:

- `pyramid.includes`
- `kotti.available_types`
- `kotti.base_includes`
- `kotti.configurators`

pyramid.includes

`pyramid.includes` defines a list of hooks that will be called when your Kotti app starts up. This gives the opportunity to third party packages to add registrations to the *Pyramid Configurator API* in order to configure views and more.

Here's an example. Let's install the kotti_twitter extension and add a Twitter profile widget to the right column of all pages. First we install the package from PyPI:

```
bin/pip install kotti_twitter
```

Then we activate the add-on in our site by editing the `pyramid.includes` setting in the `[app:kotti]` section of our INI file (if a line with `pyramid.includes` does not exist, add it).

```
pyramid.includes = kotti_twitter.include_profile_widget
```

kotti_twitter also asks us to configure the Twitter widget itself, so we add some more lines right where we were:

```
kotti_twitter.profile_widget.user = dnouri
kotti_twitter.profile_widget.loop = true
```

The order in which the includes are listed matters. For example, when you add two slots on the right hand side, the order in which you list them in `pyramid.includes` will control the order in which they will appear. As an example, here's a configuration with which the search widget will be displayed above the profile widget:

```
pyramid.includes =
    kotti_twitter.include_search_widget
    kotti_twitter.include_profile_widget
```

Read more about including packages using 'pyramid.includes' in the Pyramid documentation.

kotti.available_types

The `kotti.available_types` setting defines the list of content types available. The default configuration here is:

```
kotti.available_types = kotti.resources.Document kotti.resources.File
```

An example that removes `File` and adds two content types:

```
kotti.available_types =
    kotti.resources.Document
    kotti_calendar.resources.Calendar
    kotti_calendar.resources.Event
```

kotti.populators

The default configuration here is:

```
kotti.populators = kotti.populate.populate
```

Populators are functions with no arguments that get called on system startup. They may then make automatic changes to the database (before calling `transaction.commit()`).

kotti.search_content

Kotti provides a simple search over the content types based on kotti.resources.Content. The default configuration here is:

```
kotti.search_content = kotti.views.util.default_search_content
```

You can provide an own search function in an add-on and register this in your INI file. The return value of the search function is a list of dictionaries, each representing a search result:

```
[{'title': 'Title of search result 1',
  'description': 'Description of search result 1',
  'path': '/path/to/search-result-1'},
 {'title': 'Title of search result 2',
  'description': 'Description of search result 2',
  'path': '/path/to/search-result-2'},
 ...
 ]
```

An add-on that defines an alternative search function is kotti_solr, which provides an integration with the Solr search engine.

Configure the user interface language

By default, Kotti will display its user interface in English. The default configuration is:

```
pyramid.default_locale_name = en
```

You can configure Kotti to serve a German user interface by saying:

```
pyramid.default_locale_name = de_DE
```

The list of available languages is here.

Configure authentication and authorization

You can override the authentication and authorization policy that Kotti uses. By default, Kotti uses these factories:

```
kotti.authn_policy_factory = kotti.authtkt_factory
kotti.authz_policy_factory = kotti.acl_factory
```

These settings correspond to pyramid.authentication.AuthTktAuthenticationPolicy and pyramid.authorization.ACLAuthorizationPolicy being used.

Sessions

The `kotti.session_factory` configuration variable allows the overriding of the default session factory. By default, Kotti uses `pyramid_beaker` for sessions.

Caching

You can override Kotti's default set of cache headers by changing the `kotti.views.cache.caching_policies` dictionary, which maps policies to headers. E.g. the `Cache Resource` entry there caches all static resources for 32 days. You can also choose which responses match to which caching policy by overriding Kotti's default cache policy chooser through the use of the `kotti.caching_policy_chooser` configuration variable. The default is:

```
kotti.caching_policy_chooser = kotti.views.cache.default_caching_policy_chooser
```

URL normalization

Kotti normalizes document titles to URLs by replacing language specific characters like umlauts or accented characters with its ascii equivalents. You can change this default behavour by setting `kotti.url_normalizer.map_non_ascii_characters` configuration variable to `False`. If you do, Kotti will leave national characters in URLs.

You may also replace default component used for url normalization by setting `kotti.url_normalizer` configuation variable.

The default configuration here is:

```
kotti.url_normalzier = kotti.url_normalizer.url_normalizer
kotti.url_normalizer.map_non_ascii_characters = True
```

Local navigation

Kotti provides a build in navigation widget, which is disabled by default. To enable the navigation widget add the following to the `pyramid.includes` setting:

```
pyramid.includes = kotti.views.slots.includeme_local_navigation
```

The add-on kotti_navigation provides also a navigation widget with more features. With this add-on included your configuration looks like:

```
pyramid.includes = kotti_navigation.include_navigation_widget
```

Check the documentation of kotti_navigation for more options.

2.1.4 Automated tests

Kotti uses pytest, zope.testbrowser and WebTest for automated testing.

Before you can run the tests, you must install Kotti's 'testing' extras. Inside your Kotti checkout directory, do:

```
bin/python setup.py dev
```

To then run Kotti's test suite, do:

```
bin/py.test
```

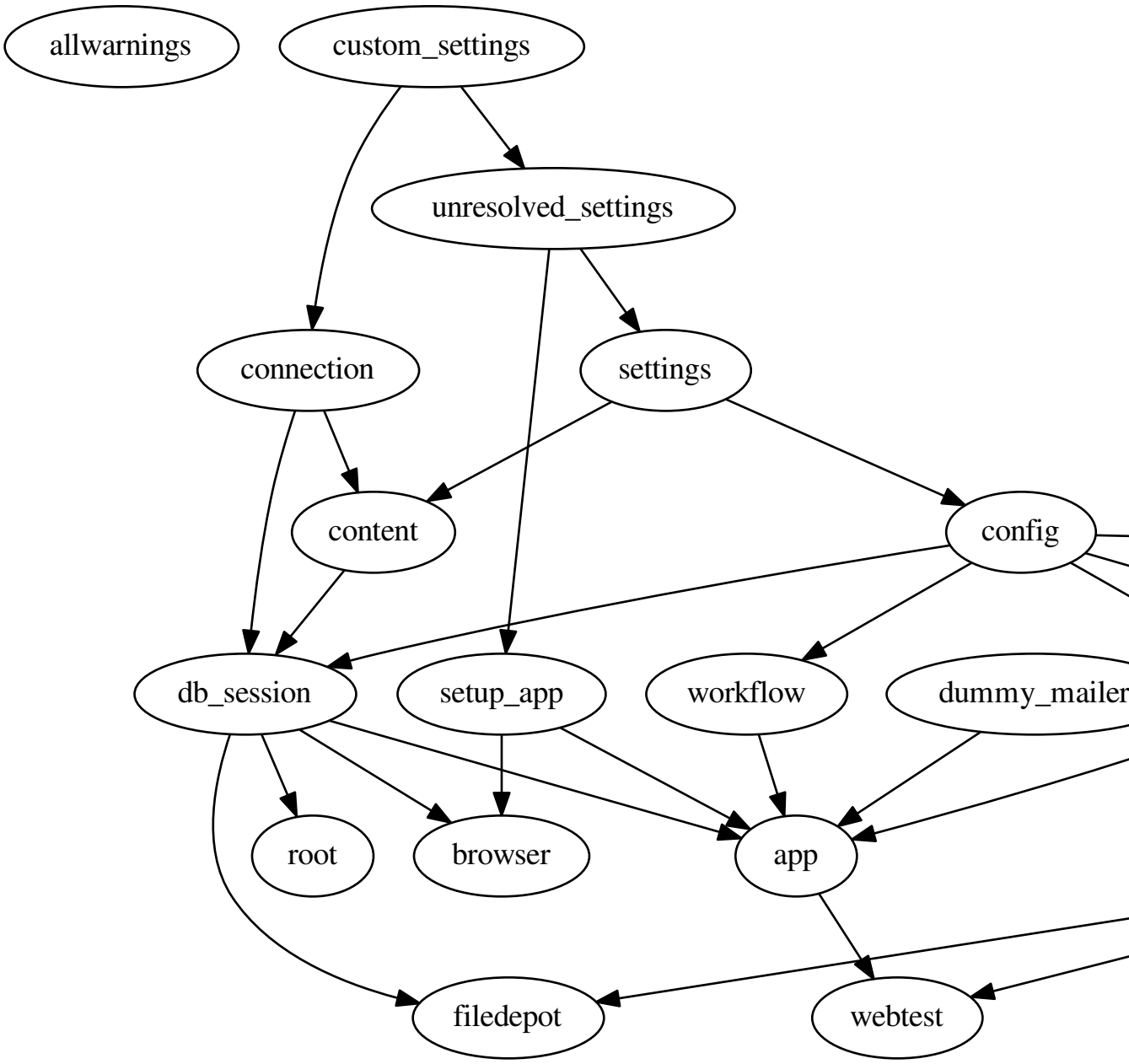
Using Kotti's test fixtures/funcargs in third party add-ons' tests

To be able to use all of Kotti's fixtures and funcargs in your own package's tests, you only need to "include" them with a line like this in your `conftest.py` file:

```
pytest_plugins = "kotti"
```

Available fixtures

Fixture dependencies



kotti.tests.**browser**(*db_session*, *request*, *setup_app*)
returns an instance of *zope.testbrowser*. The *kotti.testing.user* pytest marker (or *pytest.mark.user*) can be used

to pre-authenticate the browser with the given login name: *@user('admin')*.

kotti.tests.**config**(*request*, *settings*)
returns a Pyramid *Configurator* object initialized with Kotti's default (test) settings.

kotti.tests.**connection**(*custom_settings*)
sets up a SQLAlchemy engine and returns a connection to the database. The connection string used for testing can be specified via the `KOTTI_TEST_DB_STRING` environment variable. The `custom_settings` fixture is needed to allow users to import their models easily instead of having to override the `connection`.

kotti.tests.**content**(*connection*, *settings*)
sets up some default content using Kotti's testing populator.

kotti.tests.**custom_settings**()
This is a dummy fixture meant to be overriden in add on package's `conftest.py`. It can be used to inject arbitrary settings for third party test suites. The default settings dictionary will be updated with the dictionary returned by this fixture.

This is also a good place to import your add on's `resources` module to have the corresponding tables created during `create_all()` in *kotti.tests.content()*.

Result settings

Return type dict

kotti.tests.**db_session**(*config*, *content*, *connection*, *request*)
returns a db session object and sets up a db transaction savepoint, which will be rolled back after the test.

kotti.tests.**depot_tween**(*request*, *config*, *dummy_request*)
Sets up the Depot tween and patches Depot's `set_middleware` to suppress exceptions on subsequent calls

kotti.tests.**dummy_request**(*config*, *request*, *monkeypatch*)
returns a dummy request object after registering it as the currently active request. This is needed when *pyramid.threadlocal.get_current_request* is used.

kotti.tests.**events**(*config*, *request*)
sets up Kotti's default event handlers.

kotti.tests.**filedepot**(*db_session*, *request*, *depot_tween*)
Configures a dbsession integrated mock depot store for `depot.manager.DepotManager`

kotti.tests.**image_asset**()
Return an image file

kotti.tests.**image_asset2**()
Return another image file

kotti.tests.**mock_filedepot**(*request*, *depot_tween*)
Configures a mock depot store for `depot.manager.DepotManager`

This filedepot is not integrated with dbsession. Can be used in simple, standalone unit tests.

kotti.tests.**no_filedepots**(*db_session*, *request*, *depot_tween*)
A filedepot fixture to empty and then restore DepotManager configuration

kotti.tests.**root**(*db_session*)
returns Kotti's 'root' node.

kotti.tests.**workflow**(*config*)
loads and activates Kotti's default workflow rules.

Continuous Integration

Kotti itself is tested against Python versions 2.6 and 2.7 as well as SQLite, mySQL and PostgreSQL (in every possible combination of those) on every commit (and pull request) via the excellent GitHub / Travis CI hook.

If you want your add-on packages' to be tested the same way with additional testing against multiple versions of Kotti (including the current master), you can add a `.travis.yml` file to your repo that looks similar to this: https://raw.github.com/Kotti/kotti_media/master/.travis.yml.

The packages under http://kottipackages.xo7.de/ include all Kotti versions released on *PyPI* (synced every night at 00:15 CET) and a package built from the current master on GitHub (created every 15 minutes).

2.1.5 Translations

You can find the list of Kotti's translations here. Kotti uses GNU gettext and .po files for internationalization.

You can set the `pyramid.default_locale_name` in your configuration file to choose which language Kotti should serve the user interface (see *Configure the user interface language*).

Extraction of new messages into the `.pot` file, updating the existing `.po` files and compiling them to `.mo` files is all done with subsequent runs of the included `i18n.sh` script:

```
./i18n.sh
```

To add a new translations run:

```
./i18n.sh <2 letter code of the new language>
```

2.1.6 Deployment

Kotti deployment is not different from deploying any other WSGI app. You have a bunch of options on multiple layers: OS, RDBMS, Webserver, etc.

This document assumes the following Stack:

OS Ubuntu 12.04

Webserver Nginx

RDBMS PostgreSQL

Kotti

latest version available on PyPI
installed in its own virtualenv
deployed in an uWSGI application container

Manual installation

Install OS packages:

```
apt-get install build-essential libpq-dev python python-dev python-virtualenv
```

Install PostgreSQL:

```
apt-get install postgresql-9.1
```

Create a DB user:

```
sudo -u postgres createuser -P

Enter name of role to add: kotti
Enter password for new role:
Enter it again:
Shall the new role be a superuser? (y/n) n
Shall the new role be allowed to create databases? (y/n) n
Shall the new role be allowed to create more new roles? (y/n) n
```

Create a DB:

```
sudo -u postgres createdb -O kotti kotti
```

Install Nginx:

```
apt-get install nginx-full
```

Create a config file in `/etc/nginx/sites-available/<your_domain>.conf`:

```
server {
    listen 80;
    server_name <your_domain>;
    location / {
        include uwsgi_params;
        uwsgi_pass unix:/home/kotti/<your_domain>.sock;
    }
}
```

Create a user for your Kotti application:

```
useradd -m kotti
```

Create a virtualenv in the new user's home directory:

```
sudo -u kotti virtualenv --no-site-packages /home/kotti
```

Install Kotti and its dependencies in the virtualenv:

```
sudo -u kotti /home/kotti/bin/pip install -r https://raw.github.com/Kotti/Kotti/0.8a1/
↪requirements.txt
sudo -u kotti /home/kotti/bin/pip install Kotti==0.8a1
```

Create an ini file in `/home/kotti/kotti.ini`:

```
[app:main]
use = egg:kotti
pyramid.includes = pyramid_tm
sqlalchemy.url = postgresql://kotti:<db_password>@127.0.0.1:5432/kotti
kotti.configurators = kotti_tinymce.kotti_configure
kotti.site_title = Kotti deployed with fabric
kotti.secret = qwerty
filter-with = fanstatic

[filter:fanstatic]
use = egg:fanstatic#fanstatic
```

```
[alembic]
script_location = kotti:alembic

[uwsgi]
socket = /home/kotti/<your_domain>.sock
master = true
chmod-socket = 666
processes = 2
lazy = true      # needed if want processes > 1
lazy-apps = true
```

Install Supervisor:

```
apt-get install supervisor
```

Create a supervisor config for Kotti / uWSGI in `/etc/supervisor/conf.d/kotti.conf`:

```
[program:kotti]
autorestart=true
command=uwsgi_python --ini-paste /home/kotti/kotti.ini
directory=/home/kotti
redirect_stderr=true
```

Reload the supervisor config:

```
supervisorctl reload
```

That's all. Your Kotti deployment should now happily serve pages.

Fabfile

WARNING: this is only an example. Do not run this unmodified against a host that is intended to do anything else or things WILL break!

For your convenience there is a fabric file that automates all of the above. If you don't know what fabric is and how it works read their documentation first.

On your local machine make a separate virtualenv first and install the `fabric` and `fabtools` packages into that virtualenv:

```
mkvirtualenv kotti_deployment && cdvirtualenv
pip install fabric fabtools
```

Get the fabfile:

```
wget https://gist.github.com/gists/4079191/download
```

Read and modify the file to fit your needs. Then run it against your server:

```
fab install_all
```

You're done. Everything is installed and configured to serve Kotti under http://kotti.yourdomain.com/

2.2 Advanced Topics

2.2.1 Using Kotti as a library

Instead of taking control of your application, and delegating to your extension, you may use Kotti in applications where you define the `main` *entry point* yourself.

You'll still need to call `kotti.base_configure` from your code to set up essential parts of Kotti:

```
default_settings = {
    'pyramid.includes': 'myapp myapp.views',
    'kotti.authn_policy_factory': 'myapp.authn_policy_factory',
    'kotti.base_includes': (
        'kotti kotti.views kotti.views.login kotti.views.users'),
    'kotti.use_tables': 'orders principals',
    'kotti.populators': 'myapp.resources.populate',
    'kotti.principals_factory': 'myapp.security.Principals',
    'kotti.root_factory': 'myapp.resources.Root',
    'kotti.site_title': 'Myapp',
    }

def main(global_config, **settings):
    settings2 = default_settings.copy()
    settings2.update(settings)
    config = kotti.base_configure(global_config, **settings2)
    engine = sqlalchemy.engine_from_config(config.registry.settings, 'sqlalchemy.')
    kotti.resources.initialize_sql(engine)
    return config.make_wsgi_app()
```

The above example configures Kotti so that its user database and security subsystem are set up. Only a handful of tables (`kotti.use_tables`) and a handful of Kotti's views (`kotti.base_includes`) are activated. Furthermore, our application is configured to use a custom root factory (root node) and a custom populator.

In your *PasteDeploy* configuration you'd then wire up your app directly, maybe like this:

```
[app:myapp]
use = egg:myapp
pyramid.includes = pyramid_tm
mail.default_sender = yourname@yourhost
sqlalchemy.url = sqlite:///%(here)s/myapp.db
kotti.secret = secret

[filter:fanstatic]
use = egg:fanstatic#fanstatic

[pipeline:main]
pipeline =
    fanstatic
    myapp
```

2.2.2 Close your site to anonymous users

This recipe describes how to configure Kotti to require users to log in before they can view any of your site's pages.

To achieve this, we'll have to set our site's ACL. A custom populator will help us do that (see *kotti.populators*).

Remember that the default site ACL gives `view` privileges to every user, including anonymous (see *Security*). We'll thus have to restrict the `view` permission to the `viewer` role:

```
from kotti.resources import get_root

SITE_ACL = [
  (u'Allow', u'role:viewer', [u'view']),
  (u'Allow', u'role:editor', [u'view', u'add', u'edit']),
]

def populate():
    site = get_root()
    site.__acl__ = SITE_ACL
```

2.2.3 Default views in Kotti

In Kotti every `Content` node has a `default_view` attribute. This allows to have different views for any instance of a content type without having to append the view name to the URL.

You can also provide additional views for the default content types in your third party add on. To make them show up in the default view selector in the UI you have to append a `(view_name, view_title)` tuple to the `type_info` attribute of the respective content class via its class method `add_selectable_default_view(name, title)`.

E.g. the `kotti_media` add on provides a `media_folder_view` for the `Document` content type that lists all 'media type' children of a `Document` with their title and a media player.

Registration is done like this:

```
from kotti.resources import Document
from kotti_media import _

def includeme(config):

    Document.type_info.add_selectable_default_view("media_folder_view",
                                                   _("Media Folder"))
```

2.2.4 Adding links and actions to the edit interface

This document covers how to customize the available links and actions of the edit interface (the extra tabs and menus that appear after you log in).

The basic building block is the link, `kotti.util.Link`. Instantiate it as:

```
link = Link('name', _(u'Title'))
```

The name refers to a view name available on the context.

There's also:

- `kotti.util.LinkParent`, which allows grouping of links
- `kotti.util.LinkRenderer`, which, instead of generating a simple link, allows you to customize how it's rendered (you can insert anything there, even another submenu based on a `LinkParent`).
- `kotti.util.ActionButton`, very similar to a simple link, but generates a button instead.

Adding a new option to the Administration menu

Adding a new link as an option in the **Administration** menu, in the *Site Setup* section is easy. In your `kotti_configure` function, add:

```
from kotti.util import Link
from kotti.views.site_setup import CONTROL_PANEL_LINKS

def kotti_configure(settings):
    link = Link('name', _(u'Title'))
    CONTROL_PANEL_LINKS.append(link)
```

Make a new section in the actions menu

The *Set default view* section looks really nice. To add your own separated section in the **Action** menu and make that available to all content types:

```
from kotti.util import LinkRenderer
from kotti.resources import default_actions

def kotti_configure(settings):
    default_actions.append(LinkRenderer("my-custom-submenu"))
```

So far we've added a `LinkRenderer` to the `default_actions` which are used by all content inheriting `Content`. This LinkRenderer will render a view and insert its result in the menu.

```
@view_config(
    name="my-custom-submenu", permission="edit",
    renderer="mypackage:templates/edit/my-custom-submenu.pt")
def my_custom_submenu(context, request):
    return {}
```

And the template:

```
<tal:menu i18n:domain="mypackage">
    <li class="divider"></li>
    <li role="presentation" class="dropdown-header" i18n:translate="">
        My own actions
    </li>
    <li>
        <a i18n:translate="" href="${request.resource_url(context, 'someview')}">
            View title here
        </a>
    </li>
</tal:menu
```

2.2.5 Events

Kotti has a builtin event system that is based on the Publish-subscribe pattern.

The basic concept is that whenever a specific event occurs, all handler functions that have subscribed to that event will be executed.

There are two different types of events:

- Object events...

  ...relate to a specific object. In most cases this object will be a node from the content tree (i.e. the same as `context` in view callables).

  Events of type *ObjectEvent* have `object` and `request` attributes. `event.request` may be `None` when no request is available.

- Generic events...

  ...don't have that kind of context.

  Kotti supports such events but doesn't use them anywhere.

The event types provided by Kotti (see API docs for *kotti.events*) may be extended with your own event types. Subclass *ObjectEvent* (for object events) or `object` (for generic events) and follow the subscription instructions below, as you would for Kotti-provided events.

Subscribing to Events

To add a handler for a specific event type, you must implement a function which takes a single argument `event` and associate that to the appropriate event type by decorating it with the *subscribe* decorator.

That decorator takes up to two arguments that restrict the handler execution to specific events only. When called without arguments the handler is subscribed to *all* events:

```
from kotti.events import subscribe

@subscribe()
def all_events_handler(event):
    print event
```

To subscribe to a specific event type, supply the desired type as the first argument to *subscribe*:

```
from kotti.events import ObjectInsert
from kotti.events import subscribe

@subscribe(ObjectInsert)
def document_insert_handler(event):
    print event.object, event.request
```

You can further narrow the subscription by adding a second argument that limits the subscription to specific object types. For example, to subscribe to *ObjectDelete* events of *Document* types, write:

```
from kotti.events import ObjectDelete
from kotti.events import subscribe
from kotti.resources import Document

@subscribe(ObjectDelete, Document)
def document_delete_handler(event):
    print event.object, event.request
```

Triggering Event Handler Execution

Notifying listeners of an event is as simple as calling `notify()`:

```
from kotti.events import notify
notify(MyFunnyEvent())
```

Listeners are generally called in the order in which they are registered.

2.2.6 Use a different template for the front page (or any other page)

This recipe describes a way to override the template used for a specific object in your database. Imagine you want your front page to stand out from the rest of your site and use a unique layout.

We can set the *default view* for any content object by settings its `default_view` attribute, which is usually `None`. Inside our own populator (see *kotti.populators*), we write this:

```
from kotti.resources import get_root

def populate():
    site = get_root()
    site.default_view = 'front-page'
```

What's left is to register the `front-page` view:

```
def includeme(config):
    config.add_view(
        name='front-page',
        renderer='myapp:templates/front-page.pt',
    )
```

Note: If you want to override instead the template of *all pages*, not only that of a particluar page, you should look at the `kotti.override_assets` setting (*Override templates (kotti.asset_overrides)*).

2.2.7 Images

All image related functions were moved to kotti_image as of Kotti 1.3.0.

2.2.8 Working with Blob Data in Kotti

Kotti provides flexible mechanisms for storing and serving blob data by with the help of Depot.

Contents

How File-like Content is stored

Both `File` and `Image` store their data in `depot.fields.sqlalchemy.UploadedFileField` and they will offload their blob data to the configured depot storage. Working together with Depot configured storages means it is possible to store blob data in a variety of ways: filesystem, GridFS, Amazon storage, etc.

- `depot.io.local.LocalFileStorage`
- `depot.io.awss3.S3Storage`
- `depot.io.gridfs.GridFSStorage`
- etc.

By default Kotti will store its blob data in the configured SQL database, using *`kotti.filedepot.DBFileStorage`* storage, but you can configure your own preferred way of storing your blob data. The benefit of storing files in *`kotti.filedepot.DBFileStorage`* is having *all* content in a single place (the DB) which makes backups, exporting and importing of your site's data easy, as long as you don't have too many or too large files. The downsides of this approach appear when your database server resides on a different host (network performance becomes a greater issue) or your DB dumps become too large to be handled efficiently.

Configuration

Mountpoint

Kotti provides a Pyramid tween (pyramid.registering_tweens) that is responsible for the actual serving of blob data. It does pretty much the same as `depot.middleware.DepotMiddleware`, but is better integrated into Pyramid and therefore Kotti.

This tween "intercepts" all requests before they reach the main application (Kotti). If it's a request for blob data (identified by the configured `kotti.depot_mountpoint`), it will be served by the tween itself (or redirected to an external storage like S3), otherwise it will be "forwarded" to the main application. This mountpoint is also used to generate URLs to blobs. The default value for `kotti.depot_mountpoint` is `/depot`:

```
kotti.depot_mountpoint = /depot
```

WSGI File Wrapper

In case you have issues serving files with your WSGI server, your can try to set `kotti.depot_replace_wsgi_file_wrapper = true`. This forces Kotti to use `pyramid.response.FileIter` instead of the one provided by your WSGI server.

Storages

While Depot allows storing data in any of the configured filestorages, at this time there's no mechanism in Kotti to select, at runtime, the depot where new data will be saved. Instead, Kotti will store new files only in the configured `default` store. If, for example, you add a new depot and make that the default, you should leave the old depot configured so that Kotti will continue serving files uploaded there.

By default, Kotti comes configured with a db-based filestorage:

```
kotti.depot.0.name = dbfiles
kotti.depot.0.backend = kotti.filedepot.DBFileStorage
```

To configure a depot, several `kotti.depot.*.*` lines need to be added. The number in the first position is used to group backend configuration and to order the file storages in the configuration of Depot. The depot configured with number 0 will be the default depot, where all new blob data will be saved. There are 2 options that are required for every storage configuration: `name` and `backend`. The `name` is a unique string that will be used to identify the path of saved files (it is recorded with each blob info), so once configured for a particular storage, it should never change. The `backend` should point to a dotted path for the storage class. Any further parameters for a particular backend will be passed as keyword arguments to the backend class.

See this example, in which we store, by default, files in `/var/local/files/` using the `depot.io.local.LocalFileStorage`:

```
kotti.depot.0.name = localfs
kotti.depot.0.backend = depot.io.local.LocalFileStorage
kotti.depot.0.storage_path = /var/local/files
kotti.depot.1.name = dbfiles
kotti.depot.1.backend = kotti.filedepot.DBFileStorage
```

Notice that we kept the `dbfiles` storage, but we moved it to position 1. No blob data will be saved there anymore, but existing files in that storage will continue to be available from there.

How File-like Content is served

Starting with Kotti 1.3.0, file-like content can be served in two different ways. Let's look at an example to compare them.

Say we have a *kotti.resources.File* object in our resource tree, located at `/foo/bar/file`.

Method 1

In the default views this file is served under the URL `http://localhost/foo/bar/file/attachment-view`. This URL can be created like this:

```
>>> from kotti.resources import File
>>> file = File.query.filter(File.name == 'file').one()
>>> request.resource_url(file, 'attachment-view')
'http://localhost/foo/bar/file/attachment-view'
```

When this URL is requested, a *kotti.filedepot.StoredFileResponse* is returned:

```
>>> request.uploaded_file_response(file.data)
<StoredFileResponse at 0x10c8d22d0 200 OK>
```

The request is processed in the same way as for every other type of content in Kotti. It goes through the full traversal and view lookup machinery *with full permission checks*.

Method 2

Often these permission checks do not need to be enforced strictly. For such cases Kotti provides a "shortcut" in form of a Pyramid tween, that directly processes all requests under a certain path befor they even reach Kotti. This means: no traversal, no view lookup, no permission checks. The URL for this method can be created very similarily:

```
>>> request.uploaded_file_url(file.data, 'attachment')
'http://localhost//depot/dbfiles/68f31e97-a7f9-11e5-be07-c82a1403e6a7/download'
```

Comparison

Obviously method 2 is *a lot* faster than method 1 - typically at least by the factor of 3.

If you take a look at the callgraphs, you'll understand where this difference comes from:

Method 1	Method 2

The difference will be even more drastic, when you set up proper HTTP caching. All responses for method 2 can be cached *forever*, because the URL will change when the file's content changes.

Developing (with) File-like Content

Add a Blob Field to your Model

Adding a blob data attribute to your models can be as simple as:

```
from depot.fields.sqlalchemy import UploadedFileField
from kotti.resources import Content

class Person(Content):
    avatar = UploadedFileField()
```

While you can directly assign a `bytes` value to the `avatar` column, the `UploadedFileField` column type works best when you assign a `cgi.FieldStorage` instance as value:

```
from StringIO import StringIO
from kotti.util import _to_fieldstorage

content = '...'
data = {
        'fp': StringIO(content),
        'filename': 'avatar.png',
        'mimetype': 'image/png',
        'size': len(content),
        }
person = Person()
person.avatar = _to_fieldstorage(**data)
```

Note that the `data` dictionary described here has the same format as the deserialized value of a `deform.widget.FileUploadWidget`. See `kotti.views.edit.content.FileAddForm` and `kotti.views.edit.content.FileEditForm` for a full example of how to add or edit a model with a blob field.

Reading Blob Data

If you try directly to read data from an `UploadedFileField` you'll get a `depot.fields.upload.UploadedFile` instance, which offers a dictionary-like interface to the stored file metadata and direct access to a stream with the stored file through the `file` attribute:

```
person = DBSession.query(Person).get(1)
blob = person.avatar.file.read()
```

You should never write to the file stream directly. Instead, you should assign a new value to the `UploadedFileField` column, as described in the previous section.

Testing UploadedFileField Columns

Because `depot.manager.DepotManager` acts as a singleton, special care needs to be taken when testing features that involve saving data into `UploadedFileField` columns.

`UploadedFileField` columns require having at least one depot file storage configured. You can use a fixture called `filedepot` to have a mock file storage available for your tests.

If you're developing new depot file storages you should use the `no_filedepots` fixture, which resets the configured depots for the test run and restores the default depots back, as a teardown.

Inheritance Issues with UploadedFileField Columns

You should be aware that, presently, subclassing a model with an `UploadedFileField` column doesn't work properly. As a workaround, add a `__declare_last__` classmethod in your superclass model, similar to the one below, where we're fixing the `data` column of the `File` class.

```
from depot.fields.sqlalchemy import _SQLAMutationTracker

class File(Content):

    data = UploadedFileField()

    @classmethod
    def __declare_last__(cls):
        event.listen(cls.data, 'set', _SQLAMutationTracker._field_set, retval=True)
```

Migrating data between two different storages

Kotti provides a script that can migrate blob data from one configured stored to another and update the saved fields with the new locations. It is not needed to do this if you just want to add a new torage, or replace the default one, but you can use it if you'd like to consolidate the blob data in one place only. You can invoke the script with:

```
kotti-migrate-storage <config_uri> --from-storage <name> --to-storage <name>
```

The storage names are those assigned in the configuration file designated in `<config_uri>`. For example, let's assume you've started a website that has the default blob storage, the `DBFileStorage` named *dbfiles*. You'd like to move all the existing blob data to a `depot.io.local.LocalFileStorage` storage and make that the default. First, add the `LocalFileStorage` depot, make it the default and place the old `DBFileStorage` in position *1*::

```
kotti.depot.0.backend = depot.io.local.LocalFileStorage
kotti.depot.0.name = localfs
kotti.depot.0.storage_path = /var/local/files
kotti.depot.1.backend = kotti.filedepot.DBFileStorage
kotti.depot.1.name = dbfiles
```

Now you can invoke the migration with::

```
kotti-migrate-storage <config_uri> --from-storage dbfiles --to-storage localfs
```

As always when dealing with migrations, make sure you backup your data first!

2.2.9 Static resource management

In the default settings Kotti uses Fanstatic to manage its static resources (i.e. CSS, JS, etc.). This is accomplished by a `WSGI pipeline`:

```
[app:kotti]
use = egg:kotti

[filter:fanstatic]
use = egg:fanstatic#fanstatic

[pipeline:main]
pipeline =
    fanstatic
    kotti

[server:main]
use = egg:waitress#main
host = 127.0.0.1
port = 5000
```

Defining resources in third party addons

Defining your own resources and have them rendered in the pages produced by Kotti is also easy. You just need to define resource objects (as described in the corresponding Fanstatic documentation) and add them to either `edit_needed` or `view_needed` in kotti.fanstatic:

```
from fanstatic import Library
from fanstatic import Resource
from kotti.fanstatic import edit_needed
from kotti.fanstatic import view_needed

my_library = Library('my_package', 'resources')
my_resource = Resource(my_library, "my.js")

def includeme(config):
    # add to edit_needed if the resource is needed in edit views
    edit_needed.add(my_resource)
    # add to view_needed if the resource is needed in edit views
    view_needed.add(my_resource)
```

Don't forget to add an `entry_point` to your package's setup.py:

```
entry_points={
    'fanstatic.libraries': [
        'foo = my_package:my_library',
        ],
    },
```

Fanstatic has many more useful options, such as being able to define additional minified resources for deployment. Please consult Fanstatic's documentation for a complete list of options.

Overriding Kotti's default definitions

You can override the resources to be included in the configuration file.

The defaults are

```
[app:kotti]

kotti.fanstatic.edit_needed = kotti.fanstatic.edit_needed
kotti.fanstatic.view_needed = kotti.fanstatic.view_needed
```

which ist actually a shortcut for

```
[app:kotti]

kotti.fanstatic.edit_needed =
    kotti.fanstatic.edit_needed_js
    kotti.fanstatic.edit_needed_css

kotti.fanstatic.view_needed =
    kotti.fanstatic.view_needed_js
    kotti.fanstatic.view_needed_css
```

You may add as many `kotti.fanstatic.NeededGroup`, `fanstatic.Group` or `fanstatic.Resource` (or actually anything that provides a `.need()` method) objects in dotted notation as you want.

Say you want to completely abandon Kotti's CSS resources (and use your own for both view and edit views) but use Kotti's JS resources plus an additional JS resource defined within your app (only in edit views). Your configuration file might look like this:

```
[app:kotti]

kotti.fanstatic.edit_needed =
    kotti.fanstatic.edit_needed_js
    myapp.fanstatic.js_resource
    myapp.fanstatic.css_resource

kotti.fanstatic.view_needed =
    kotti.fanstatic.view_needed_js
    myapp.fanstatic.css_resource
```

Using Kotti without Fanstatic

To handle resources yourself, you can easily and completely turn off fanstatic:

```
[app:main]
use = egg:kotti

[server:main]
use = egg:waitress#main
host = 127.0.0.1
port = 5000
```

2.2.10 Understanding Kotti's startup phase

1. When a Kotti application is started the `kotti.main()` function is called by the WSGI server and is passed a `settings` dictionary that contains all key / value pairs from the `[app:kotti]` section of the `*.ini` file.
2. The `settings` dictionary is passed to `kotti.base_configure()`. This is where the main work happens:
   (a) Every key in *kotti.conf_defaults* that is not in the `settings` dictionary (i.e. that is not in the `.ini` file) is copied to the `settings` dictionary, together with the default value for that key.
   (b) Add-on initializations: all functions that are listed in the `kotti.configurators` parameter are resolved and called.
   (c) `pyramid.includes` are removed from the `settings` dictionary for later processing, i.e. **after** `kotti.base_includes`.
   (d) A class:*pyramid.config.Configurator* is instanciated with the remaining `settings`.
   (e) The `kotti.base_includes` (containing various Kotti subsystems, such as `kotti.events`, `kotti.views`, etc.) are passed to `config.include`.
   (f) The `pyramid.includes` that were removed from the `settings` dictionary in step 2.3 are processed.
   (g) The `kotti.zcml_includes` are processed.
3. The SQLAlchemy engine is created with the connection URL that is defined in the sqlalchemy.url parameter in the `.ini` file.
4. The fully configured WSGI application is returned to the WSGI server and is ready to process requests.

2.2.11 Sanitizers

Kotti provides a mechanism to *sanitize* arbitrary strings.

You can configure *available* sanitizers via `kotti.sanitizers`. This setting takes a list of strings, with each specifying a `name:callable` pair. `name` is the name under which this sanitizer is registered. `callable` is a dotted path to a function taking an unsanitized string and returning a sanitized version of it.

The default configuration is:

```
kotti.sanitizers =
    xss_protection:kotti.sanitizers.xss_protection
    minimal_html:kotti.sanitizers.minimal_html
    no_html:kotti.sanitizers.no_html
```

For thorough explaination of the included sanitizers see *kotti.sanitizers*.

Explicit sanitization

You can explicitly use any configured sanitizer like this:

```
from kotti.sanitizers import sanitize

sanitzed = sanitize(unsanitized, 'xss_protection')
```

The sanitize function is also available as a method of the *kotti.views.util.TemplateAPI*. This is just a convenience wrapper to ease usage in templates:

```
${api.sanitize(context.foo, 'minimal_html')}
```

Sanitize on write (implicit sanitization)

The second setting related to sanitization is `kotti.sanitize_on_write`. It defines, for the specified resource classes, the attributes that are sanitized and the sanitizers that will be used when the attributes are mutated and flushed.

This setting takes a list of `dotted_path:sanitizer_name(s)` pairs. `dotted_path` is a dotted path to a resource class attribute that will be sanitized implicitly with the respective sanitizer(s) upon write access. `sanitizer_name(s)` is a comma separated list of available sanitizer names as configured above.

Kotti will setup *listeners* for the *kotti.events.ObjectInsert* and *kotti.events.ObjectUpdate* events for the given classes and attach a function that filters the respective attributes with the specified sanitizer.

This means that *any* write access to configured attributes through your application (also within correctly setup command line scripts) will be sanitized *implicitly*.

The default configuration is:

```
kotti.sanitize_on_write =
    kotti.resources.Document.body:xss_protection
    kotti.resources.Content.title:no_html
```

You can also use multiple sanitizers:

```
kotti.sanitize_on_write =
    kotti.resources.Document.body:xss_protection,some_other_sanitizer
```

Implementing a custom sanitizer

A sanitizer is just a function that takes and returns a string. It can be as simple as:

```
def no_dogs_allowed(html):
    return html.replace('dogs', 'cats')

no_dogs_allowed('<p>I love dogs.</p>')
... '<p>I love cats.</p>'
```

You can also look at *kotti.sanitizers* for examples.

CHAPTER 3

API

3.1 API Documentation

kotti.**includeme**(*config*)
Pyramid includeme hook.

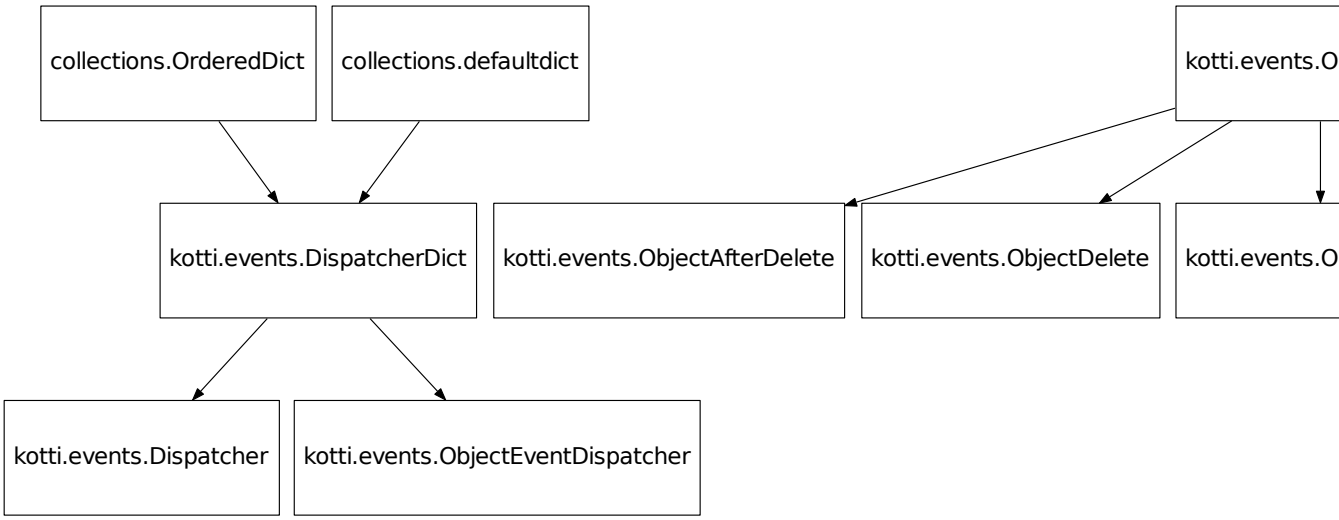
Parameters **config** (pyramid.config.Configurator) – app config

3.1.1 kotti.events

This module includes a simple events system that allows users to subscribe to specific events, and more particularly to *object events* of specific object types.

See also: *Events*.

Inheritance Diagram



class kotti.events.**ObjectEvent**(*obj*, *request=None*)
Event related to an object.

class kotti.events.**ObjectInsert**(*obj*, *request=None*)
This event is emitted when an object is inserted into the DB.

class kotti.events.**ObjectUpdate**(*obj*, *request=None*)
This event is emitted when an object in the DB is updated.

class kotti.events.**ObjectDelete**(*obj*, *request=None*)
This event is emitted when an object is deleted from the DB.

class kotti.events.**ObjectAfterDelete**(*obj*, *request=None*)
This event is emitted after an object has been deleted from the DB.

Deprecated since version 0.9.

class kotti.events.**UserDeleted**(*obj*, *request=None*)
This event is emitted when an user object is deleted from the DB.

class kotti.events.**DispatcherDict**(**args*, ***kwargs*)
Base class for dispatchers

class kotti.events.**Dispatcher**(**args*, ***kwargs*)
Dispatches based on event type.

```
>>> class BaseEvent(object): pass
>>> class SubEvent(BaseEvent): pass
>>> class UnrelatedEvent(object): pass
>>> def base_listener(event):
...     print('Called base listener')
>>> def sub_listener(event):
```

```
...     print('Called sub listener')
>>> def unrelated_listener(event):
...     print('Called unrelated listener')
...     return 1
```

```
>>> dispatcher = Dispatcher()
>>> dispatcher[BaseEvent].append(base_listener)
>>> dispatcher[SubEvent].append(sub_listener)
>>> dispatcher[UnrelatedEvent].append(unrelated_listener)
```

```
>>> dispatcher(BaseEvent())
Called base listener
[None]
>>> dispatcher(SubEvent())
Called base listener
Called sub listener
[None, None]
>>> dispatcher(UnrelatedEvent())
Called unrelated listener
[1]
```

class kotti.events.**ObjectEventDispatcher**(**args*, ***kwargs*)

Dispatches based on both event type and object type.

```
>>> class BaseObject(object): pass
>>> class SubObject(BaseObject): pass
>>> def base_listener(event):
...     return 'base'
>>> def subobj_insert_listener(event):
...     return 'sub'
>>> def all_listener(event):
...     return 'all'
```

```
>>> dispatcher = ObjectEventDispatcher()
>>> dispatcher[(ObjectEvent, BaseObject)].append(base_listener)
>>> dispatcher[(ObjectInsert, SubObject)].append(subobj_insert_listener)
>>> dispatcher[(ObjectEvent, None)].append(all_listener)
```

```
>>> dispatcher(ObjectEvent(BaseObject()))
['base', 'all']
>>> dispatcher(ObjectInsert(BaseObject()))
['base', 'all']
>>> dispatcher(ObjectEvent(SubObject()))
['base', 'all']
>>> dispatcher(ObjectInsert(SubObject()))
['base', 'sub', 'all']
```

kotti.events.**set_owner**(*event*)

Set `owner` of the object that triggered the event.

Parameters **event** (*ObjectInsert*) – event that trigerred this handler.

kotti.events.**set_creation_date**(*event*)

Set `creation_date` of the object that triggered the event.

Parameters **event** (*ObjectInsert*) – event that trigerred this handler.

kotti.events.**set_modification_date**(*event*)
Update modification_date of the object that triggered the event.

Parameters **event** (*ObjectUpdate*) – event that trigerred this handler.

kotti.events.**delete_orphaned_tags**(*event*)
Delete Tag instances / records when they are not associated with any content.

Parameters **event** (*ObjectAfterDelete*) – event that trigerred this handler.

kotti.events.**cleanup_user_groups**(*event*)
Remove a deleted group from the groups of a user/group and remove all local group entries of it.

Parameters **event** (*UserDeleted*) – event that trigerred this handler.

kotti.events.**reset_content_owner**(*event*)
Reset the owner of the content from the deleted owner.

Parameters **event** (*UserDeleted*) – event that trigerred this handler.

class kotti.events.**subscribe**(*evttype=<type 'object'>*, *objtype=None*)
Function decorator to attach the decorated function as a handler for a Kotti event. Example:

```
from kotti.events import ObjectInsert
from kotti.events import subscribe
from kotti.resources import Document

@subscribe()
def on_all_events(event):
    # this will be executed on *every* event
    print "Some kind of event occured"

@subscribe(ObjectInsert)
def on_insert(event):
    # this will be executed on every object insert
    context = event.object
    request = event.request
    print "Object insert"

@subscribe(ObjectInsert, Document)
def on_document_insert(event):
    # this will only be executed on object inserts if the object is
    # is an instance of Document
    context = event.object
    request = event.request
    print "Document insert"
```

kotti.events.**wire_sqlalchemy**()
Connect SQLAlchemy events to their respective handler function (that fires the corresponding Kotti event).

kotti.events.**includeme**(*config*)
Pyramid includeme hook.

Parameters **config** (pyramid.config.Configurator) – app config

3.1.2 kotti.fanstatic

class kotti.fanstatic.**NeededGroup**(*resources=None*)
A collection of fanstatic resources that supports dynamic appending of resources after initialization

add(*resource*)
resource may be a:

- `fanstatic.Resource` object or
- `fanstatic.Group` object

3.1.3 kotti.interfaces

interface kotti.interfaces.**INode**
Extends: `pyramid.interfaces.ILocation`

Marker interface for all nodes (and subclasses)

interface kotti.interfaces.**IContent**
Extends: *kotti.interfaces.INode*

Marker interface for all nodes of type Content (and subclasses thereof)

interface kotti.interfaces.**IDocument**
Extends: *kotti.interfaces.IContent*

Marker interface for all nodes of type Document (and subclasses thereof)

interface kotti.interfaces.**IFile**
Extends: *kotti.interfaces.IContent*

Marker interface for all nodes of type File (and subclasses thereof)

interface kotti.interfaces.**IDefaultWorkflow**
Marker interface for content classes that want to use the default workflow

interface kotti.interfaces.**INavigationRoot**
Marker interface for content nodes / classes that want to be the root for the navigation.

Considering a content tree like this:

```
- /a
  - /a/a
  - /a/b (provides INavigationRoot)
    - /a/b/a
    - /a/b/b
    - /a/b/c
  - a/c

The root item for the navigation will be ``/a/b`` for everey context in
or below ``/a/b`` and ``/a`` for every other item.
```

3.1.4 kotti.message

kotti.message.**validate_token**(*user*, *token*, *valid_hrs=24*)

```
>>> from kotti.testing import setUp, tearDown
>>> ignore = setUp()
>>> class User(object):
...     pass
>>> daniel = User()
>>> daniel.name = u'daniel'
```

```
>>> alice = User()
>>> alice.name = u'alice'
>>> token = make_token(daniel)
>>> validate_token(daniel, token)
True
>>> validate_token(alice, token)
False
>>> validate_token(daniel, 'foo')
False
>>> token = make_token(daniel, seconds=time.time() - 100000)
>>> validate_token(daniel, token)
False
>>> validate_token(daniel, token, valid_hrs=48)
True
>>> tearDown()
```

kotti.message.**send_email**(*request*, *recipients*, *template_name*, *template_vars=None*)
General email sender.

Parameters

- **request** (*kotti.request.Request*) – current request.
- **recipients** (*list*) – list of email addresses. Each email should be a string like: u'"John Doe" <joedoe@foo.com>'.
- **template_name** (*string*) – asset specification (e.g. 'mypackage:templates/email.pt')
- **template_vars** (*dict*) – set of variables present on template.

3.1.5 kotti.migrate

This module aims to make it easier to run the Alembic migration scripts of Kotti and Kotti add-ons by providing a uniform access.

Commands herein will typically be called by the console script `kotti-migrate` (see the docstring of that command below).

Kotti stores the current revision of its migration in table `kotti_alembic_versions`. The convention here is `<packagename>_alembic_versions`. You should normally not need to worry about the name of this table, as it is created and managed automatically through this module. If, however, you plan to use your own alembic.ini configuration file for your add-on or application, keep in mind to configure a table name as described above. The table name can be set using Alembic's `version_table` option.

Kotti has start-up code that will create the database from scratch if it doesn't exist. This code will also call this module's function `stamp_heads` to set the current revision of all migrations registered with this module to the latest. This assumes that when we create the database from scratch (using `metadata.create_all`), we don't need to run any of the past migrations.

Unfortunately, this won't help in the situation where a user adds an add-on with migrations to the Kotti site _after_ the database was initialized for the first time. In this case, users of the add-on will need to run `kotti-migrate stamp_head --scripts=yourpackage:alembic`, or the add-on author will have to write equivalent code somewhere in their populate hook.

Add-on authors can register their Alembic scripts with this module by adding their Alembic 'script directory' location to the `kotti.alembic_dirs` setting. An example:

```
def kotti_configure(settings):
    # ...
    settings['kotti.alembic_dirs'] += ' kotti_contactform:alembic'
```

`kotti-migrate` commands 'list_all', 'upgrade_all' and 'stamp_heads' will then include the add-on.

3.1.6 kotti.populate

Populate contains two functions that are called on application startup (if you haven't modified kotti.populators).

`kotti.populate.`**`populate_users`**`()`
Create the admin user with the password from the `kotti.secret` option if there is no user with name 'admin' yet.

`kotti.populate.`**`populate`**`()`
Create the root node (*Document*) and the 'about' subnode in the nodes tree if there are no nodes yet.

3.1.7 kotti.request

class `kotti.request.`**Request**(*environ*, *charset=None*, *unicode_errors=None*, *decode_param_names=None*, ***kw*)
Bases: `pyramid.request.Request`

Kotti subclasses `pyramid.request.Request` to make additional attributes / methods available on request objects and override Pyramid's `pyramid.request.Request.has_permission()`. The latter is needed to support Kotti's concept of local roles not just for users but also for groups (*kotti.security.list_groups_callback()*).

user
Add the authenticated user to the request object.

Result the currently authenticated user

Return type *kotti.security.Principal* or whatever is returned by the custom principals database defined in the `kotti.principals_factory` setting

has_permission(*permission*, *context=None*)
Check if the current request has the given permission on the current or explicitly passed context. This is different from `pyramid.request.Request.has_permission`` ()` in that a context other than the one bound to the request can be passed. This allows to consider local roles for the check.

Parameters

- **permission** (*str*) – name of the permission to check
- **context** (*kotti.resources.Node*) – context for which the permission is checked. Defaults to the context on which the request invoked.

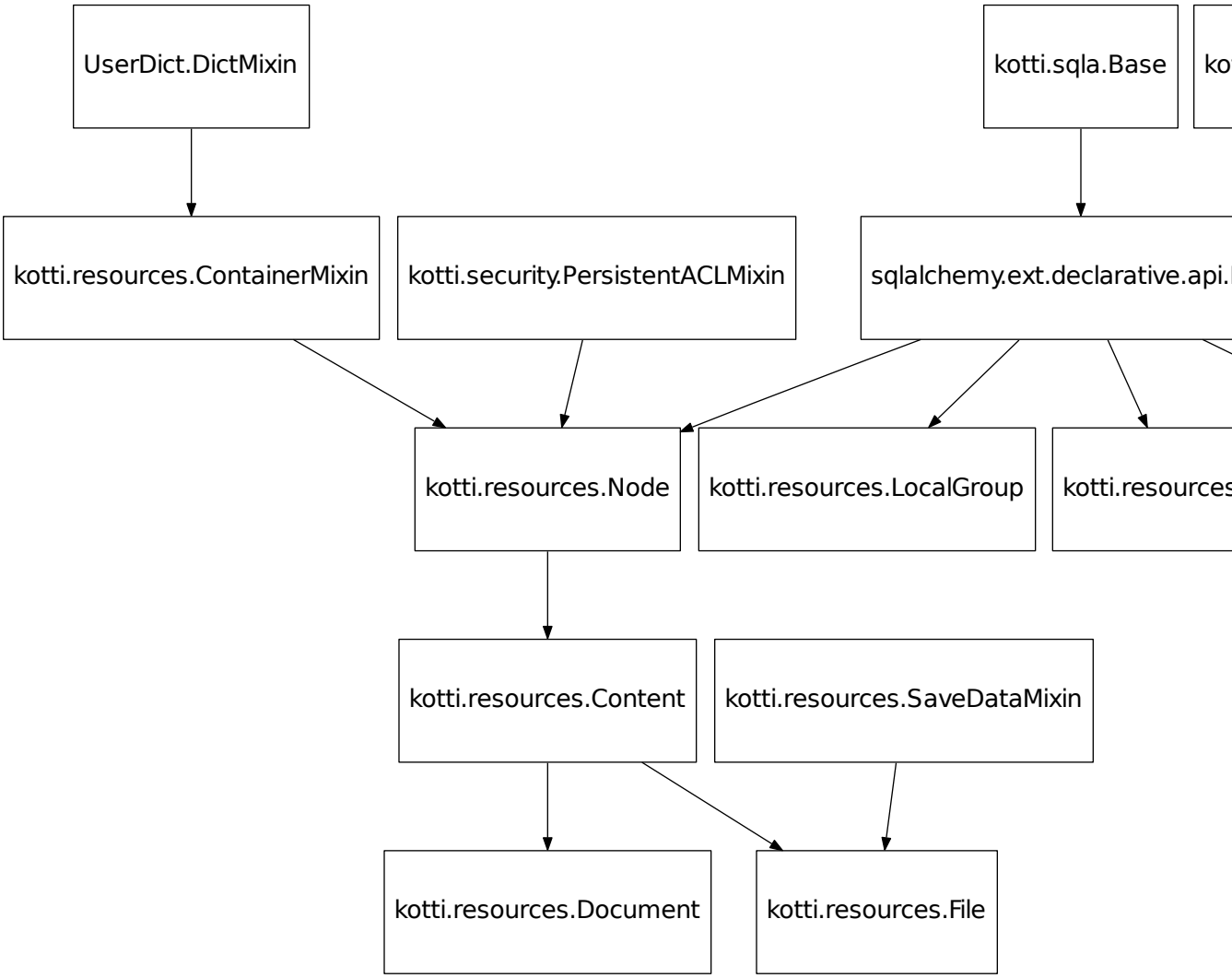
Result True if has_permission, False else

Return type bool

3.1.8 kotti.resources

The *resources* module contains all the classes for Kotti's persistence layer, which is based on SQLAlchemy.

Inheritance Diagram



class kotti.resources.**ContainerMixin**

Bases: object, UserDict.DictMixin

Containers form the API of a Node that's used for subitem access and in traversal.

keys()

Result children names

Return type list

children_with_permission(*request*, *permission='view'*)

Return only those children for which the user initiating the request has the asked permission.

Parameters

- **request** (*kotti.request.Request*) – current request
- **permission** (*str*) – The permission for which you want the allowed children

Result List of child nodes

Return type list

class kotti.resources.**LocalGroup**(*node*, *principal_name*, *group_name*)

Bases: sqlalchemy.ext.declarative.api.Base

Local groups allow the assignment of groups or roles to principals (users or groups) **for a certain context** (i.e. a *Node* in the content tree).

id
Primary key for the node in the DB (sqlalchemy.types.Integer)

node_id
ID of the node for this assignment (sqlalchemy.types.Integer)

principal_name
Name of the principal (user or group) (sqlalchemy.types.Unicode)

group_name
Name of the assigned group or role (sqlalchemy.types.Unicode)

class kotti.resources.**Node**(*name=None*, *parent=None*, *title=u''*, *annotations=None*, ***kwargs*)

Bases: sqlalchemy.ext.declarative.api.Base, *kotti.resources.ContainerMixin*, kotti.security.PersistentACLMixin

Basic node in the persistance hierarchy.

id
Primary key for the node in the DB (sqlalchemy.types.Integer)

type
Lowercase class name of the node instance (sqlalchemy.types.String)

parent_id
ID of the node's parent (sqlalchemy.types.Integer)

position
Position of the node within its container / parent (sqlalchemy.types.Integer)

path
The path can be used to efficiently filter for child objects (sqlalchemy.types.Unicode).

name
Name of the node as used in the URL (sqlalchemy.types.Unicode)

title
Title of the node, e.g. as shown in search results (sqlalchemy.types.Unicode)

annotations
Annotations can be used to store arbitrary data in a nested dictionary (kotti.sqla.NestedMustationDict)

copy(***kwargs*)

Result A copy of the current instance

Return type *Node*

class kotti.resources.**TypeInfo**(***kwargs*)
Bases: object

TypeInfo instances contain information about the type of a node.

You can pass arbitrary keyword arguments in the constructor, they will become instance attributes. The most common are:

- name
- title
- add_view
- addable_to
- edit_links
- selectable_default_views
- uploadable_mimetypes
- add_permission

copy(***kwargs*)

Result a copy of the current TypeInfo instance

Return type *TypeInfo*

addable(*context*, *request*)

Parameters

- **context** (Content or subclass thereof (or anything that has a type_info attribute of type *TypeInfo*)) –
- **request** (*kotti.request.Request*) – current request

Result True if the type described in 'self' may be added to 'context', False otherwise.

Return type Boolean

add_selectable_default_view(*name*, *title*)
Add a view to the list of default views selectable by the user in the UI.

Parameters

- **name** (*str*) – Name the view is registered with
- **title** (*unicode or TranslationString*) – Title for the view for display in the UI.

is_uploadable_mimetype(*mimetype*)
Check if uploads of the given MIME type are allowed.

Parameters **mimetype** (*str*) – MIME type

Result Upload allowed (>0) or forbidden (0). The greater the result, the better is the match. E.g. `image/*` (6) is a better match for `image/png` than * (1).

Return type int

class kotti.resources.**Tag**(***kwargs*)
Bases: sqlalchemy.ext.declarative.api.Base

Basic tag implementation. Instances of this class are just the tag itself and can be mapped to instances of *Content* (or any of its descendants) via instances of *TagsToContents*.

id
Primary key column in the DB (sqlalchemy.types.Integer)

title
Title of the tag sqlalchemy.types.Unicode

items

Result

Return type list

class kotti.resources.**TagsToContents**(***kwargs*)
Bases: sqlalchemy.ext.declarative.api.Base

Tags to contents mapping

tag_id
Foreign key referencing *Tag.id* (sqlalchemy.types.Integer)

content_id
Foreign key referencing *Content.id* (sqlalchemy.types.Integer)

tag
Relation that adds a content_tags sqlalchemy.orm.backref() to *Tag* instances to allow easy access to all content tagged with that tag. (sqlalchemy.orm.relationship())

position
Ordering position of the tag sqlalchemy.types.Integer

title
title of the associated *Tag* instance (sqlalchemy.ext.associationproxy.association_proxy)

class kotti.resources.**Content**(*name=None*, *parent=None*, *title=u''*, *annotations=None*, *default_view=None*, *description=u''*, *language=None*, *owner=None*, *creation_date=None*, *modification_date=None*, *in_navigation=True*, *tags=None*, ***kwargs*)
Bases: *kotti.resources.Node*

Content adds some attributes to *Node* that are useful for content objects in a CMS.

id
Primary key column in the DB (sqlalchemy.types.Integer)

state
Workflow state of the content object (sqlalchemy.types.String)

type_info = <kotti.resources.TypeInfo object>
type_info is a class attribute (*TypeInfo*)

default_view
Name of the view that should be displayed to the user when visiting an URL without a explicit view name appended (sqlalchemy.types.String)

description
Description of the content object. In default Kotti this is used e.g. in the description tag in the HTML, in the search results and rendered below the title in most views. (sqlalchemy.types.Unicode)

language
Language code (ISO 639) of the content object (sqlalchemy.types.Unicode)

owner
Owner (username) of the content object (sqlalchemy.types.Unicode)

in_navigation
Shall the content be visible in the navigation? (sqlalchemy.types.Boolean)

creation_date
Date / time the content was created (sqlalchemy.types.DateTime)

modification_date
Date / time the content was last modified (sqlalchemy.types.DateTime)

tags
Tags assigned to the content object (list of str)

class kotti.resources.**Document**(*body=u''*, *mime_type='text/html'*, ***kwargs*)
Bases: *kotti.resources.Content*

Document extends *Content* with a body and its mime_type. In addition Document and its descendants implement *IDefaultWorkflow* and therefore are associated with the default workflow (at least in unmodified Kotti installations).

id
Primary key column in the DB (sqlalchemy.types.Integer)

type_info = <kotti.resources.TypeInfo object>
type_info is a class attribute (*TypeInfo*)

body
Body text of the Document (sqlalchemy.types.Unicode)

mime_type
MIME type of the Document (sqlalchemy.types.String)

class kotti.resources.**SaveDataMixin**(*data=None*, *filename=None*, *mimetype=None*, *size=None*, ***kwargs*)
Bases: object

The classmethods must not be implemented on a class that inherits from Base with SQLAlchemy>=1.0, otherwise that class cannot be subclassed further.

See http://stackoverflow.com/questions/30433960/how-to-use-declare-last-in-sqlalchemy-1-0 # noqa

classmethod from_field_storage(*fs*)

Create and return an instance of this class from a file upload through a webbrowser.

Parameters **fs** (cgi.FieldStorage) – FieldStorage instance as found in a *kotti.request.Request*'s POST MultiDict.

Result The created instance.

Return type *kotti.resources.File*

filename = Column(None, Unicode(length=100), table=None)
The filename is used in the attachment view to give downloads the original filename it had when it was uploaded. (sqlalchemy.types.Unicode)

mimetype = Column(None, String(length=100), table=None)
MIME type of the file (sqlalchemy.types.String)

size = Column(None, Integer(), table=None)
Size of the file in bytes (sqlalchemy.types.Integer)

copy(***kwargs*)
Same as *Content.copy* with additional data support. data needs some special attention, because we don't want the same depot file to be assigned to multiple content nodes.

class kotti.resources.**File**(*data=None*, *filename=None*, *mimetype=None*, *size=None*, ***kwargs*)
Bases: *kotti.resources.SaveDataMixin*, *kotti.resources.Content*

File adds some attributes to *Content* that are useful for storing binary data.

id
Primary key column in the DB (sqlalchemy.types.Integer)

kotti.resources.**get_root**(*request=None*)

Call the function defined by the kotti.root_factory setting and return its result.

Parameters **request** (*kotti.request.Request*) – current request (optional)

Result a node in the node tree

Return type *Node* or descendant;

class kotti.resources.**DefaultRootCache**
Bases: object

Default implementation for *get_root()*

root_id
Query for the one node without a parent and return its id. :result: The root node's id. :rtype: int

get_root()
Query for the root node by its id. This enables SQLAlchemy's session cache (query is executed only once per session). :result: The root node. :rtype: *Node*.

3.1.9 kotti.filedepot

class kotti.filedepot.**DBFileStorage**
Implementation of depot.io.interfaces.FileStorage,

Uses *kotti.filedepot.DBStoredFile* to store blob data in an SQL database.

create(*content*, *filename=None*, *content_type=None*)
Saves a new file and returns the file id

Parameters

- **content** – can either be bytes, another file object or a cgi.FieldStorage. When filename and content_type parameters are not provided they are deducted from the content itself.
- **filename** (*string*) – filename for this file
- **content_type** (*string*) – Mimetype of this file

Returns the unique file_id associated to this file

Return type string

delete(*file_or_id*)
Deletes a file. If the file didn't exist it will just do nothing.

Parameters **file_or_id** – can be either DBStoredFile or a file_id

exists(*file_or_id*)
Returns if a file or its ID still exist.

Returns Returns if a file or its ID still exist.

Return type bool

static get(*file_id*)
Returns the file given by the file_id

Parameters **file_id** (*string*) – the unique id associated to the file

Result a *kotti.filedepot.DBStoredFile* instance

Return type *kotti.filedepot.DBStoredFile*

replace(*file_or_id*, *content*, *filename=None*, *content_type=None*)
Replaces an existing file, an `IOError` is raised if the file didn't already exist.

Given a `StoredFile` or its ID it will replace the current content with the provided `content` value. If `filename` and `content_type` are provided or can be deducted by the `content` itself they will also replace the previous values, otherwise the current values are kept.

Parameters

- **file_or_id** – can be either `DBStoredFile` or a `file_id`
- **content** – can either be `bytes`, another `file object` or a `cgi.FieldStorage`. When `filename` and `content_type` parameters are not provided they are deducted from the content itself.
- **filename** (*string*) – filename for this file
- **content_type** (*string*) – Mimetype of this file

class kotti.filedepot.**DBStoredFile**(*file_id*, *filename=None*, *content_type=None*, *last_modified=None*, *content_length=None*, ***kwds*)
`depot.io.interfaces.StoredFile` implementation that stores file data in SQL database.

Can be used together with *kotti.filedepot.DBFileStorage* to implement blobs storage in the database.

static close(**args*, ***kwargs*)
Implement `StoredFile.close()`. *DBStoredFile* never closes.

static closed()
Implement `StoredFile.closed()`.

content_length
Size of the blob in bytes (`sqlalchemy.types.Integer`)

content_type
MIME type of the blob (`sqlalchemy.types.String`)

data
The binary data itself (`sqlalchemy.types.LargeBinary`)

file_id
Unique file id given to this blob (`sqlalchemy.types.String`)

filename
The original filename it had when it was uploaded. (`sqlalchemy.types.String`)

id
Primary key column in the DB (`sqlalchemy.types.Integer`)

last_modified
Date / time the blob was created or last modified (`sqlalchemy.types.DateTime`)

name
Implement `StoredFile.name()`.

Result the filename of the saved file

Return type string

read(*n=-1*)
Reads `n` bytes from the file.

If `n` is not specified or is `-1` the whole file content is read in memory and returned

seek(*offset*, *whence=0*)
Change stream position.

Change the stream position to the given byte offset. The offset is interpreted relative to the position indicated by whence.

Parameters

- **offset** (*int*) – Position for the cursor
- **whence** (*int*) –
  - **0 – start of stream (the default);** offset should be zero or positive
  - 1 – current stream position; offset may be negative
  - 2 – end of stream; offset is usually negative

static seekable()
Implement `StoredFile.seekable()`.

tell()
Returns current position of file cursor

Result Current file cursor position.

Return type int

static writable()
Implement `StoredFile.writable()`.

class kotti.filedepot.**StoredFileResponse**(*f*, *request*, *disposition='attachment'*, *cache_max_age=604800*, *content_type=None*, *content_encoding=None*)
A Response object that can be used to serve an UploadedFile instance.

Code adapted from `pyramid.response.FileResponse`.

class kotti.filedepot.**TweenFactory**(*handler*, *registry*)
Factory for a Pyramid tween in charge of serving Depot files.

This is the Pyramid tween version of `depot.middleware.DepotMiddleware`. It does exactly the same as Depot's WSGI middleware, but operates on a `pyramid.request.Request` object instead of the WSGI environment.

kotti.filedepot.**extract_depot_settings**(*prefix='kotti.depot.'*, *settings=None*)
Merges items from a dictionary that have keys that start with *prefix* to a list of dictionaries.

Parameters

- **prefix** (*string*) – A dotted string representing the prefix for the common values
- **settings** – A dictionary with settings. Result is extracted from this

kotti.filedepot.**includeme**(*config*)
Pyramid includeme hook.

Parameters **config** (`pyramid.config.Configurator`) – app config

kotti.filedepot.**set_metadata**(*event*)
Set DBStoredFile metadata based on data

Parameters **event** (ObjectInsert or ObjectUpdate) – event that trigerred this handler.

3.1.10 kotti.sanitizers

For a high level introduction and available configuration options see *Sanitizers*.

kotti.sanitizers.**sanitize**(*html*, *sanitizer*)
Sanitize HTML

Parameters

- **html** (*basestring*) – HTML to be sanitized
- **sanitizer** (*str*) – name of the sanitizer to use

Result sanitized HTML

Return type unicode

kotti.sanitizers.**xss_protection**(*html*)
Sanitizer that removes tags that are not considered XSS safe. See bleach_whitelist.generally_xss_unsafe for a complete list of tags that are removed. Attributes and styles are left untouched.

Parameters **html** (*basestring*) – HTML to be sanitized

Result sanitized HTML

Return type unicode

kotti.sanitizers.**minimal_html**(*html*)
Sanitizer that only leaves a basic set of tags and attributes. See bleach_whitelist.markdown_tags, bleach_whitelist.print_tags, bleach_whitelist.markdown_attrs, bleach_whitelist.print_attrs for a complete list of tags and attributes that are allowed. All styles are completely removed.

Parameters **html** (*basestring*) – HTML to be sanitized

Result sanitized HTML

Return type unicode

kotti.sanitizers.**no_html**(*html*)
Sanitizer that removes **all** tags.

Parameters **html** (*basestring*) – HTML to be sanitized

Result plain text

Return type unicode

kotti.sanitizers.**includeme**(*config*)
Pyramid includeme hook.

Parameters **config** (pyramid.config.Configurator) – app config

3.1.11 kotti.security

kotti.security.**has_permission**(*permission*, *context*, *request*)
Check if the current request has a permission on the given context.

Deprecated since version 0.9.

Parameters

- **permission** (*str*) – permission to check for
- **context** (:class:kotti.resources.Node) – context that should be checked for the given permission
- **request** (*kotti.request.Request*) – current request

Result True if request has the permission, False else

Return type bool

class kotti.security.**Principal**(*name*, *password=None*, *active=True*, *confirm_token=None*, *title=u''*, *email=None*, *groups=None*)
A minimal 'Principal' implementation.

The attributes on this object correspond to what one ought to implement to get full support by the system. You're free to add additional attributes.

- As convenience, when passing 'password' in the initializer, it is hashed using 'get_principals().hash_password'
- The boolean 'active' attribute defines whether a principal may log in. This allows the deactivation of accounts without deleting them.
- The 'confirm_token' attribute is set whenever a user has forgotten their password. This token is used to identify the receiver of the email. This attribute should be set to 'None' once confirmation has succeeded.

class kotti.security.**AbstractPrincipals**
This class serves as documentation and defines what methods are expected from a Principals database.

Principals mostly provides dict-like access to the principal objects in the database. In addition, there's the 'search' method which allows searching users and groups.

'hash_password' is for initial hashing of a clear text password, while 'validate_password' is used by the login to see if the entered password matches the hashed password that's already in the database.

Use the 'kotti.principals' settings variable to override Kotti's default Principals implementation with your own.

keys()
Return a list of principal ids that are in the database.

search(***kwargs*)
Return an iterable with principal objects that correspond to the search arguments passed in.

This example would return all principals with the id 'bob':

```
get_principals().search(name=u'bob')
```

Here, we ask for all principals that have 'bob' in either their 'name' or their 'title'. We pass '*bob*' instead of 'bob' to indicate that we want case-insensitive substring matching:

```
get_principals().search(name=u'bob', title=u'bob')
```

This call should fail with AttributeError unless there's a 'foo' attribute on principal objects that supports search:

```
get_principals().search(name=u'bob', foo=u'bar')
```

hash_password(*password*)
Return a hash of the given password.

This is what's stored in the database as 'principal.password'.

validate_password(*clear*, *hashed*)
Returns True if the clear text password matches the hash.

kotti.security.**list_groups**(*name*, *context=None*)
List groups for principal with a given `name`.

The optional `context` argument may be passed to check the list of groups in a given context.

kotti.security.**set_groups**(*name*, *context*, *groups_to_set=()*)
Set the list of groups for principal with given `name` and in given `context`.

kotti.security.**list_groups_callback**(*name*, *request*)
List the groups for the principal identified by `name`. Consider `authz_context` to support assigment of local roles to groups.

kotti.security.**principals_with_local_roles**(*context*, *inherit=True*)
Return a list of principal names that have local roles in the context.

class kotti.security.**Principals**
Kotti's default principal database.

Look at 'AbstractPrincipals' for documentation.

This is a default implementation that may be replaced by using the 'kotti.principals' settings variable.

factory
alias of *Principal*

search(*match='any'*, ***kwargs*)
Search the principal database.

Parameters

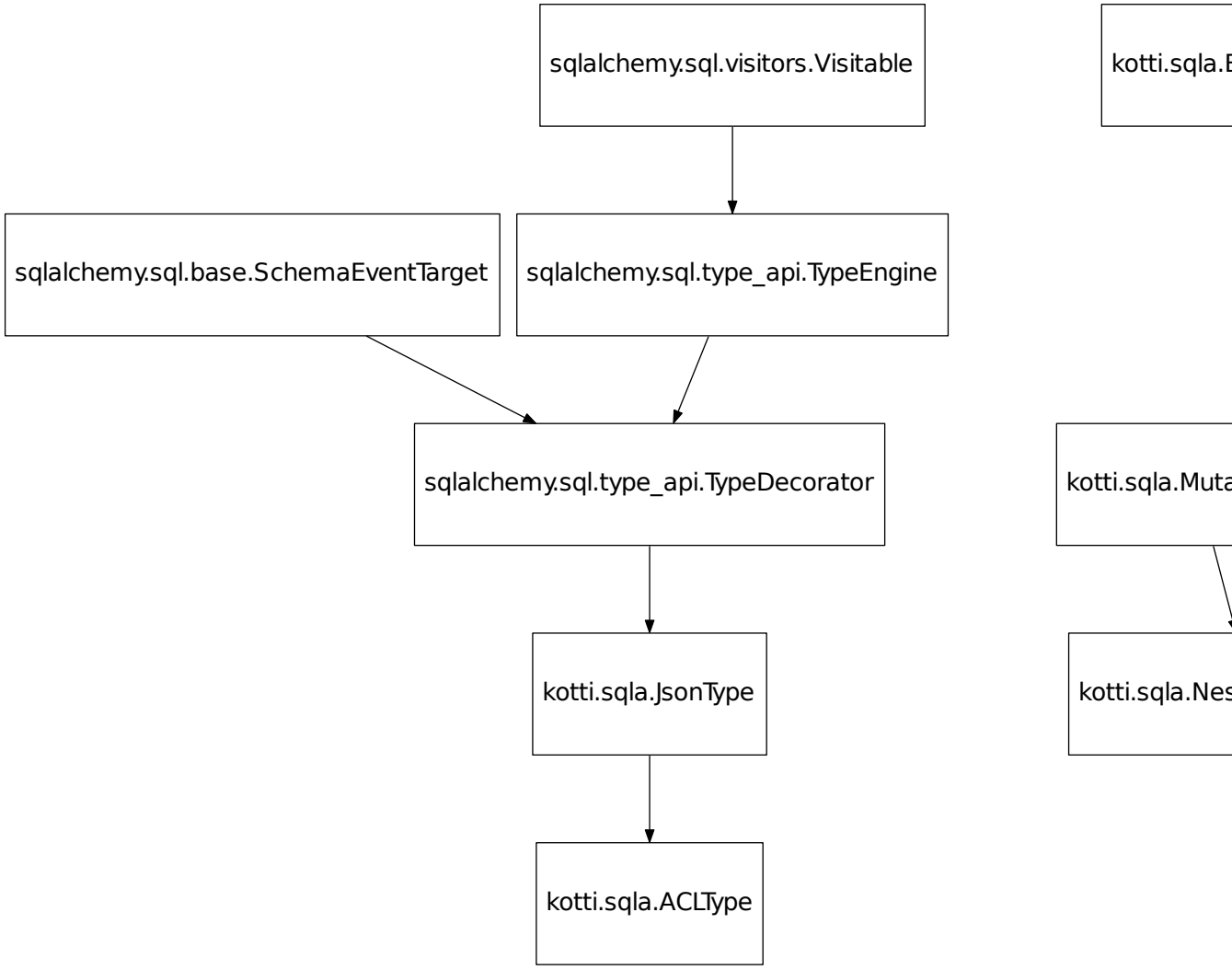
- **match** (*str*) – `any` to return all principals matching any search param, `all` to return only principals matching all params
- **kwargs** (*varying.*) – Search conditions, e.g. `name='bob', active=True`.

Result SQLAlchemy query object

Return type `sqlalchemy.orm.query.Query`

3.1.12 kotti.sqla

Inheritance Diagram



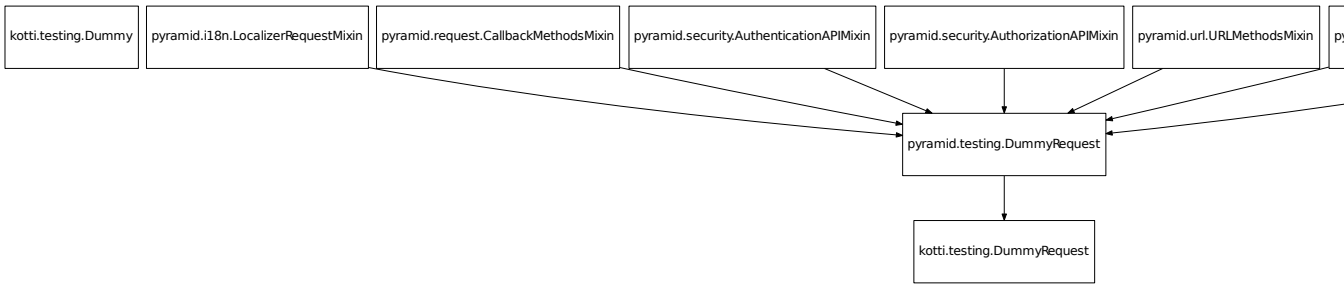
class kotti.sqla.**JsonType**(**args*, ***kwargs*)
http://www.sqlalchemy.org/docs/core/types.html#marshal-json-strings

impl
alias of Text

class kotti.sqla.**MutationDict**(*data*)
http://www.sqlalchemy.org/docs/orm/extensions/mutable.html

3.1.13 kotti.testing

Inheritance Diagram



kotti.testing.**includeme_login**(*config*)
Pyramid includeme hook.

Parameters **config** (pyramid.config.Configurator) – app config

kotti.testing.**includeme_layout**(*config*)
Pyramid includeme hook.

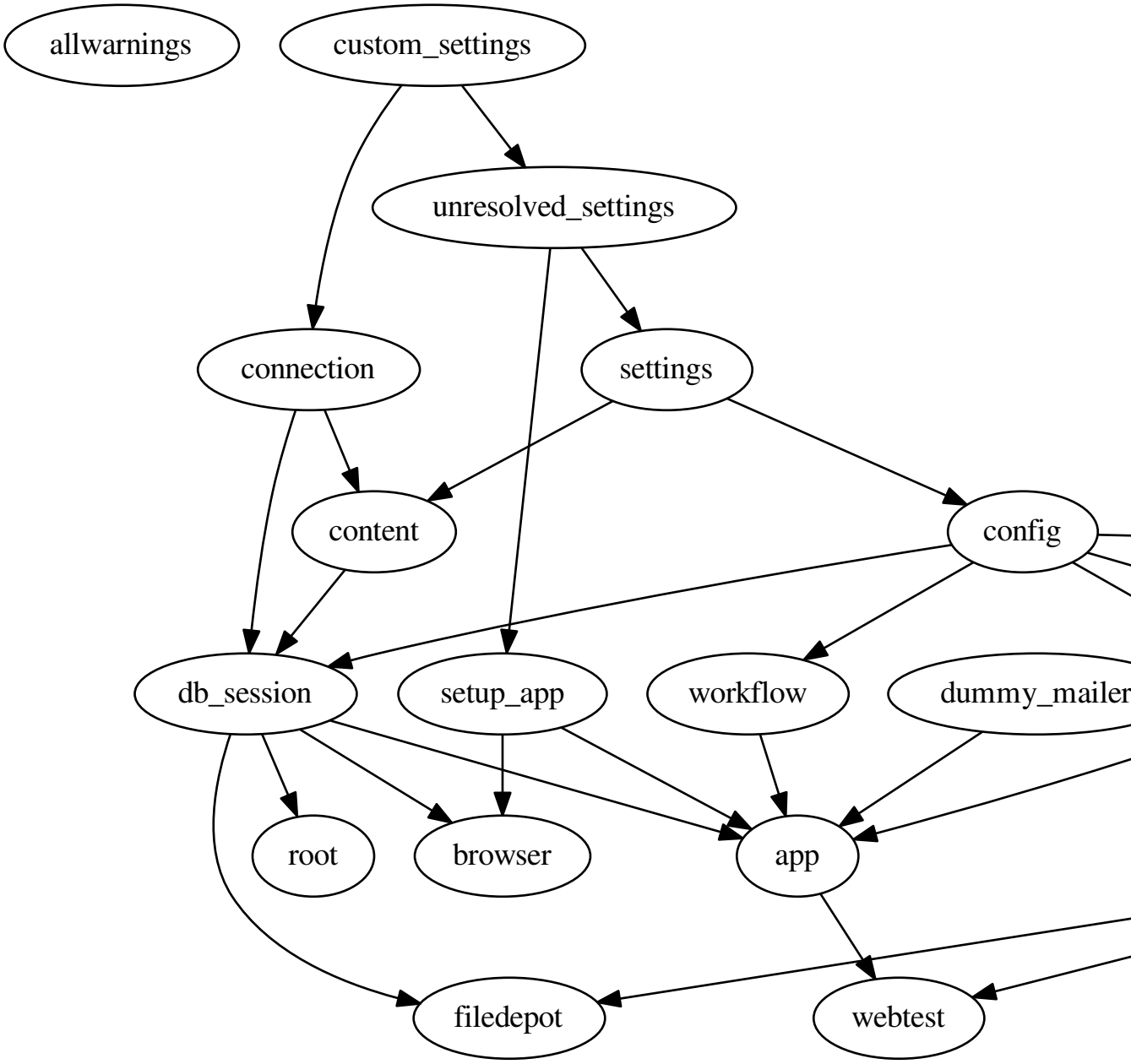
Parameters **config** (pyramid.config.Configurator) – app config

kotti.testing.**include_testing_view**(*config*)
Pyramid includeme hook.

Parameters **config** (pyramid.config.Configurator) – app config

3.1.14 kotti.tests

Fixture dependencies



kotti.tests.**image_asset**()
Return an image file

kotti.tests.**image_asset2**()
Return another image file

kotti.tests.**custom_settings**()
This is a dummy fixture meant to be overriden in add on package's `conftest.py`. It can be used to inject arbitrary settings for third party test suites. The default settings dictionary will be updated with the dictionary returned by this fixture.

This is also a good place to import your add on's `resources` module to have the corresponding tables created during `create_all()` in *kotti.tests.content()*.

Result settings

Return type dict

kotti.tests.**config**(*request*, *settings*)
returns a Pyramid *Configurator* object initialized with Kotti's default (test) settings.

kotti.tests.**connection**(*custom_settings*)
sets up a SQLAlchemy engine and returns a connection to the database. The connection string used for testing can be specified via the `KOTTI_TEST_DB_STRING` environment variable. The `custom_settings` fixture is needed to allow users to import their models easily instead of having to override the `connection`.

kotti.tests.**content**(*connection*, *settings*)
sets up some default content using Kotti's testing populator.

kotti.tests.**db_session**(*config*, *content*, *connection*, *request*)
returns a db session object and sets up a db transaction savepoint, which will be rolled back after the test.

kotti.tests.**dummy_request**(*config*, *request*, *monkeypatch*)
returns a dummy request object after registering it as the currently active request. This is needed when *pyramid.threadlocal.get_current_request* is used.

kotti.tests.**events**(*config*, *request*)
sets up Kotti's default event handlers.

kotti.tests.**browser**(*db_session*, *request*, *setup_app*)
returns an instance of *zope.testbrowser*. The *kotti.testing.user* pytest marker (or *pytest.mark.user*) can be used to pre-authenticate the browser with the given login name: *@user('admin')*.

kotti.tests.**root**(*db_session*)
returns Kotti's 'root' node.

kotti.tests.**workflow**(*config*)
loads and activates Kotti's default workflow rules.

kotti.tests.**depot_tween**(*request*, *config*, *dummy_request*)
Sets up the Depot tween and patches Depot's `set_middleware` to suppress exceptions on subsequent calls

kotti.tests.**mock_filedepot**(*request*, *depot_tween*)
Configures a mock depot store for `depot.manager.DepotManager`

This filedepot is not integrated with dbsession. Can be used in simple, standalone unit tests.

kotti.tests.**filedepot**(*db_session*, *request*, *depot_tween*)
Configures a dbsession integrated mock depot store for `depot.manager.DepotManager`

kotti.tests.**no_filedepots**(*db_session*, *request*, *depot_tween*)
A filedepot fixture to empty and then restore DepotManager configuration

3.1.15 kotti.traversal

This module contains Kotti's node tree traverser.

In Kotti versions < 1.3.0, Pyramid's default traverser (`pyramid.traversal.ResourceTreeTraverser`) was used. This traverser still works, but it becomes decreasingly performant the deeper your resource tree is nested. This is caused by the fact, that it generates one DB query per level, whereas the Kotti traverser (*`kotti.traversal.NodeTreeTraverser`*) generates a single DB query, regardless of the number of request path segments. This query not only finds the context, but also returns all node items in its lineage. This means, that neither accessing `context.parent` nor calling `pyramid.location.lineage()` will result in additional DB queries.

The performance benefits are huge. The table below compares the requests per seconds (rps) that were reached on a developer's notebook against a PostgreSQL database with 4419 *`kotti.resources.Document`* nodes.

request.path	Pyramid traverser (rps)	Kotti traverser (rps)
/	49	49
/a/	41	36
/a/b/	30	35
/a/b/c/	23	34
/a/b/c/d/	19	33
/a/b/c/d/e/	16	33
/a/b/c/d/e/f/	14	33
/a/b/c/d/e/f/g/	12	32
/a/b/c/d/e/f/g/h/	11	31
/a/b/c/d/e/f/g/h/i/	10	30
/a/b/c/d/e/f/g/h/i/j/	8	29

class kotti.traversal.**NodeTreeTraverser**(*root*)

An optimized resource tree traverser for *`kotti.resources.Node`* based resource trees.

static traverse(*root*, *vpath_tuple*)

Parameters

- **root** (*`kotti.resources.Node`*) – The node where traversal should start
- **vpath_tuple** (*`tuple`*) – Tuple of path segments to be traversed

Returns List of nodes, from root (excluded) to context (included). Each node has its parent set already, so that no subsequent queries will be be performed, e.g. when calling `lineage(context)`

Return type list of *`kotti.resources.Node`*

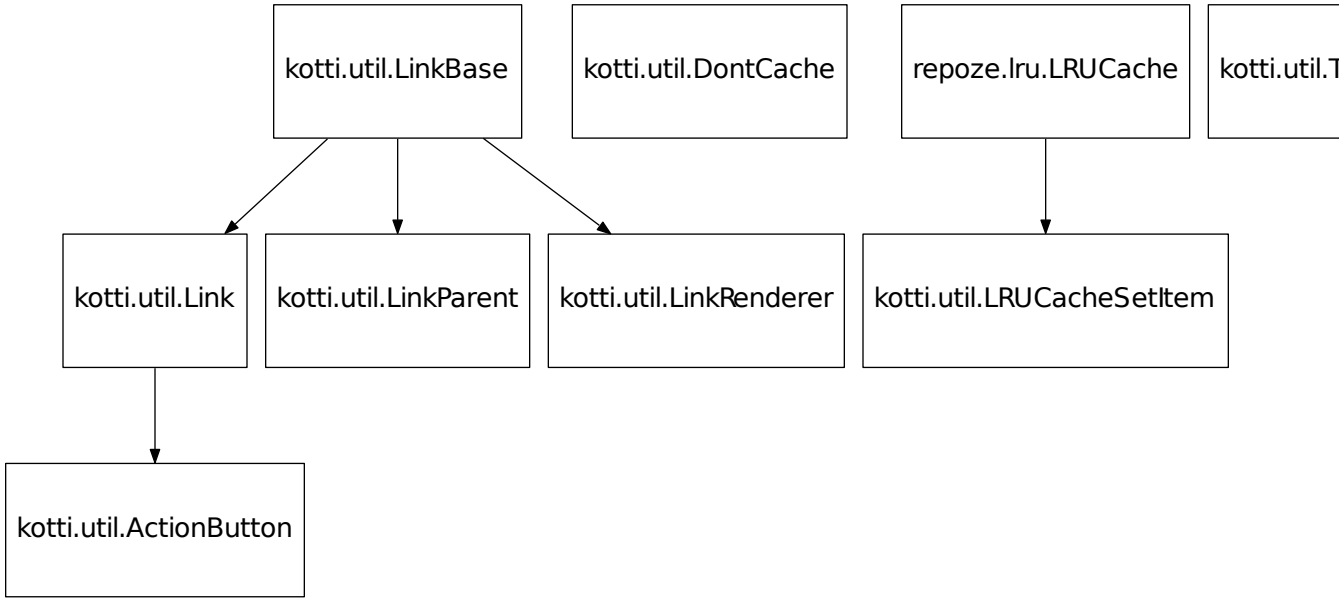
kotti.traversal.**includeme**(*config*)

Pyramid includeme hook.

Parameters **config** (`pyramid.config.Configurator`) – app config

3.1.16 kotti.util

Inheritance Diagram



class kotti.util.**LinkRenderer**(*name*, *predicate=None*)
Bases: kotti.util.LinkBase

A menu link that renders a view to render the link.

class kotti.util.**LinkParent**(*title*, *children*)
Bases: kotti.util.LinkBase

A menu link that renders sublinks in a dropdown.

kotti.util.**extract_from_settings**(*prefix*, *settings=None*)

```
>>> settings = {
...     'kotti_twitter.foo_bar': '1', 'kotti.spam_eggs': '2'}
>>> print(extract_from_settings('kotti_twitter.', settings))
{'foo_bar': '1'}
```

kotti.util.**title_to_name**(*title*, *blacklist=()*, *max_length=None*)
If max_length is None, fallback to the name column size (*kotti.resources.Node*)

kotti.util.**camel_case_to_name**(*text*)

```
>>> camel_case_to_name('FooBar')
'foo_bar'
```

```
>>> camel_case_to_name('TXTFile')
'txt_file'
>>> camel_case_to_name ('MyTXTFile')
'my_txt_file'
>>> camel_case_to_name('froBOZ')
'fro_boz'
>>> camel_case_to_name('f')
'f'
```

3.1.17 kotti.views

class kotti.views.**BaseView**(*context*, *request*)
Very basic view class that can be subclassed. Does nothing more than assignment of `context` and `request` to instance attributes on initialization.

kotti.views.**includeme**(*config*)
Pyramid includeme hook.

Parameters **config** (pyramid.config.Configurator) – app config

kotti.views.cache

kotti.views.cache.**set_max_age**(*response*, *delta*, *cache_ctrl=None*)
Sets max-age and expires headers based on the timedelta *delta*.

If *cache_ctrl* is not None, I'll add items found therein to the Cache-Control header.

Will overwrite existing values and preserve non overwritten ones.

kotti.views.cache.**includeme**(*config*)
Pyramid includeme hook.

Parameters **config** (pyramid.config.Configurator) – app config

kotti.views.edit

Edit views.

kotti.views.edit.**includeme**(*config*)
Pyramid includeme hook.

Parameters **config** (pyramid.config.Configurator) – app config

kotti.views.edit.actions

Action views

class kotti.views.edit.actions.**NodeActions**(*context*, *request*)
Bases: object

Actions related to content nodes.

back(*view=None*)
Redirect to the given view of the context, the referrer of the request or the default_view of the context.

Return type pyramid.httpexceptions.HTTPFound

workflow_change()
Handle workflow change requests from workflow dropdown.

Result Redirect response to the referrer of the request.

Return type pyramid.httpexceptions.HTTPFound

copy_node()
Copy nodes view. Copy the current node or the selected nodes in the contents view and save the result in the session of the request.

Result Redirect response to the referrer of the request.

Return type pyramid.httpexceptions.HTTPFound

cut_nodes()
Cut nodes view. Cut the current node or the selected nodes in the contents view and save the result in the session of the request.

Result Redirect response to the referrer of the request.

Return type pyramid.httpexceptions.HTTPFound

paste_nodes()
Paste nodes view. Paste formerly copied or cutted nodes into the current context. Note that a cutted node can not be pasted into itself.

Result Redirect response to the referrer of the request.

Return type pyramid.httpexceptions.HTTPFound

move(*move*)
Do the real work to move the selected nodes up or down. Called by the up and the down view.

Result Redirect response to the referrer of the request.

Return type pyramid.httpexceptions.HTTPFound

up()
Move up nodes view. Move the selected nodes up by 1 position and get back to the referrer of the request.

Result Redirect response to the referrer of the request.

Return type pyramid.httpexceptions.HTTPFound

down()
Move down nodes view. Move the selected nodes down by 1 position and get back to the referrer of the request.

Result Redirect response to the referrer of the request.

Return type pyramid.httpexceptions.HTTPFound

set_visibility(*show*)
Do the real work to set the visibility of nodes in the menu. Called by the show and the hide view.

Result Redirect response to the referrer of the request.

Return type pyramid.httpexceptions.HTTPFound

show()
Show nodes view. Switch the in_navigation attribute of selected nodes to `True` and get back to the referrer of the request.

Result Redirect response to the referrer of the request.

Return type pyramid.httpexceptions.HTTPFound

hide()
Hide nodes view. Switch the in_navigation attribute of selected nodes to `False` and get back to the referrer of the request.

Result Redirect response to the referrer of the request.

Return type pyramid.httpexceptions.HTTPFound

delete_node()
Delete node view. Renders either a view to delete the current node or handle the deletion of the current node and get back to the default view of the node.

Result Either a redirect response or a dictionary passed to the template for rendering.

Return type pyramid.httpexceptions.HTTPFound or dict

delete_nodes()
Delete nodes view. Renders either a view to delete multiple nodes or delete the selected nodes and get back to the referrer of the request.

Result Either a redirect response or a dictionary passed to the template for rendering.

Return type pyramid.httpexceptions.HTTPFound or dict

rename_node()
Rename node view. Renders either a view to change the title and name for the current node or handle the changes and get back to the default view of the node.

Result Either a redirect response or a dictionary passed to the template for rendering.

Return type pyramid.httpexceptions.HTTPFound or dict

rename_nodes()
Rename nodes view. Renders either a view to change the titles and names for multiple nodes or handle the changes and get back to the referrer of the request.

Result Either a redirect response or a dictionary passed to the template for rendering.

Return type pyramid.httpexceptions.HTTPFound or dict

change_state()
Change state view. Renders either a view to handle workflow changes for multiple nodes or handle the selected workflow changes and get back to the referrer of the request.

Result Either a redirect response or a dictionary passed to the template for rendering.

Return type pyramid.httpexceptions.HTTPFound or dict

kotti.views.edit.actions.**contents_buttons**(*context*, *request*)
Build the action buttons for the contents view based on the current state and the persmissions of the user.

Result List of ActionButtons.

Return type list

kotti.views.edit.actions.**content_type_factories**(*context*, *request*)
Renders the drop down menu for Add button in editor bar.

Result Dictionary passed to the template for rendering.

Return type pyramid.httpexceptions.HTTPFound or dict

kotti.views.edit.actions.**contents**(*context*, *request*)
Contents view. Renders either the contents view or handle the action button actions of the view.

Result Either a redirect response or a dictionary passed to the template for rendering.

Return type pyramid.httpexceptions.HTTPFound or dict

kotti.views.edit.actions.**move_child_position**(*context*, *request*)
Move the child from one position to another.

Parameters

- **context** (*:class:kotti.resources.Node or descendant*) – "Container" node in which the child changes its position.
- **request** – Current request (of method POST). Must contain either "from" and "to" params or a json_body that contain(s) the 0-based old (i.e. the current index of the child to be moved) and new position (its new index) values.

Result JSON serializable object with a single attribute ("result") that is either "success" or "error".

Return type dict

kotti.views.edit.actions.**workflow**(*context*, *request*)
Renders the drop down menu for workflow actions.

Result Dictionary passed to the template for rendering.

Return type dict

kotti.views.edit.actions.**actions**(*context*, *request*)
Renders the drop down menu for Actions button in editor bar.

Result Dictionary passed to the template for rendering.

Return type dict

kotti.views.edit.actions.**includeme**(*config*)
Pyramid includeme hook.

Parameters **config** (pyramid.config.Configurator) – app config

kotti.views.edit.content

Content edit views

kotti.views.edit.content.**includeme**(*config*)
Pyramid includeme hook.

Parameters **config** (pyramid.config.Configurator) – app config

kotti.views.edit.default_views

summary Default view selctor views

kotti.views.edit.default_views.**includeme**(*config*)
Pyramid includeme hook.

Parameters **config** (pyramid.config.Configurator) – app config

kotti.views.file

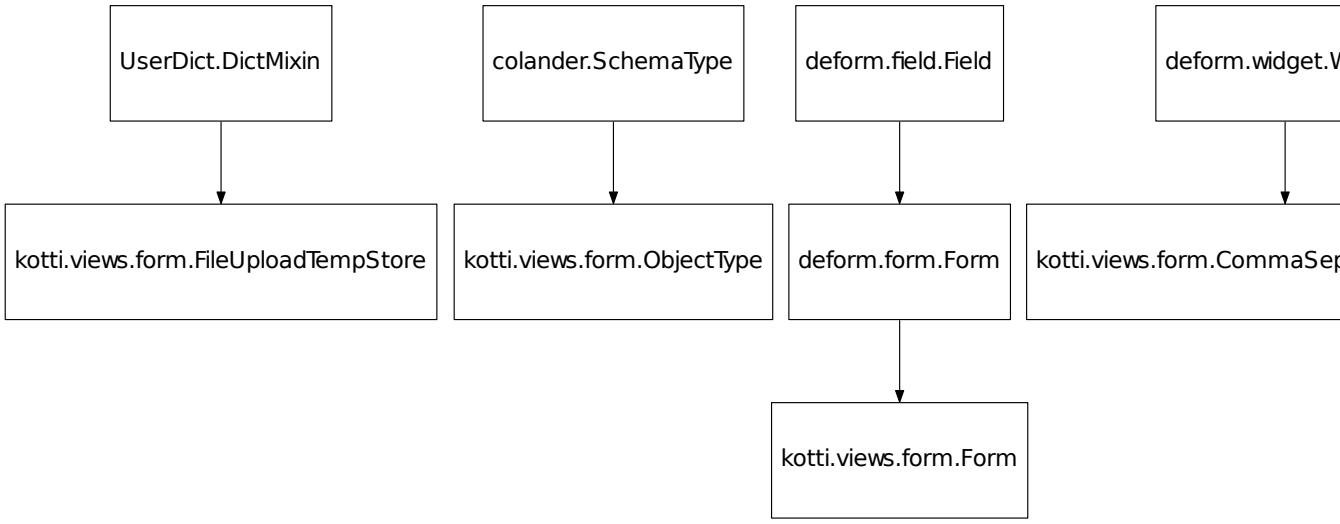
kotti.views.file.**includeme**(*config*)
Pyramid includeme hook.

Parameters **config** (pyramid.config.Configurator) – app config

kotti.views.form

Form related base views from which you can inherit.

Inheritance Diagram



class kotti.views.form.**ObjectType**
A type leaving the value untouched.

class kotti.views.form.**Form**(*schema*, *action=''*, *method='POST'*, *buttons=()*, *formid='deform'*, *use_ajax=False*, *ajax_options='{}'*, *autocomplete=None*, ***kw*)
A deform Form that allows 'appstruct' to be set on the instance.

class kotti.views.form.**BaseFormView**(*context*, *request*, ***kwargs*)
A basic view for forms with save and cancel buttons.

form_class
alias of *Form*

class kotti.views.form.**EditFormView**(*context*, *request*, ***kwargs*)
A base form for content editing purposes.

Set *self.schema_factory* to the context's schema. Values of fields in this schema will be set as attributes on the context. An example:

```
import colander
from deform.widget import RichTextWidget

from kotti.edit.content import ContentSchema
from kotti.edit.content import EditFormView

class DocumentSchema(ContentSchema):
```

```
    body = colander.SchemaNode(
        colander.String(),
        title=u'Body',
        widget=RichTextWidget(),
        missing=u'',
        )

class DocumentEditForm(EditFormView):
    schema_factory = DocumentSchema
```

class kotti.views.form.**AddFormView**(*context*, *request*, ***kwargs*)
A base form for content adding purposes.

Set *self.schema_factory* as with EditFormView. Also set *item_type* to your model class. An example:

```
class DocumentAddForm(AddFormView):
    schema_factory = DocumentSchema
    add = Document
    item_type = u'Document'
```

class kotti.views.form.**FileUploadTempStore**(*request*)
A temporary storage for file file uploads

File uploads are stored in the session so that you don't need to upload your file again if validation of another schema node fails.

kotti.views.form.**validate_file_size_limit**(*node*, *value*)
File size limit validator.

You can configure the maximum size by setting the kotti.max_file_size option to the maximum number of bytes that you want to allow.

kotti.views.login

Login / logout and forbidden views and forms.

class kotti.views.login.**UserSelfRegistered**(*obj*, *request=None*)
This event is emitted just after user self registered. Intended use is to allow addons to do some preparation for such user - create custom contents, nodes etc. Event handler object parameter is a Principal object

kotti.views.login.**login_success_callback**(*request*, *user*, *came_from*)
Default implementation of kotti.login_success_callback. You can implement a custom function with the same signature and point the kotti.login_success_callback setting to it.

Parameters

- **request** (*kotti.request.Request*) – Current request
- **user** (kotti.security.Princial) – Principal, who just logged in successfully.
- **came_from** (*str*) – URL the user came from

Result Any Pyramid response object, by default a redirect to came_from or the context where login was called.

Return type pyramid.httpexceptions.HTTPFound

kotti.views.login.**reset_password_callback**(*request*, *user*)
Default implementation of kotti.reset_password_callback. You can implement a custom function with the same signature and point the kotti.reset_password_callback setting to it.

Parameters

- **request** (*kotti.request.Request*) – Current request
- **user** (kotti.security.Princial) – Principal, who's password was requested to be reset.

Result Any Pyramid response object, by default a redirect to to the same URL from where the password reset was called.

Return type pyramid.httpexceptions.HTTPFound

kotti.views.login.**login**(*context*, *request*)

Login view. Renders either the login or password forgot form templates or handles their form submission and redirects to came_from on success.

Result Either a redirect response or a dictionary passed to the template for rendering

Return type pyramid.httpexceptions.HTTPFound or dict

kotti.views.login.**logout**(*context*, *request*)

Logout view. Always redirects the user to where he came from.

Result Redirect to came_from

Return type pyramid.httpexceptions.HTTPFound

class kotti.views.login.**SetPasswordSchema**(**arg*, ***kw*)

Schema for the set password form

password = None

colander.String

token = None

colander.String

email = None

colander.String

continue_to = None

colander.String

kotti.views.login.**set_password**(*context*, *request*, *success_msg=u'You have reset your password.'*)

Set password view. Displays the set password form and handles its form submission.

Parameters

- **context** (*kotti.resources.Content*) – Current context
- **request** (*kotti.request.Request*) – Current request
- **success_msg** (*str or TranslationString*) – Message to display on successful submission handling

Result Redirect response or dictionary passed to the template for rendering.

Return type pyramid.httpexceptions.HTTPFound or dict

kotti.views.login.**forbidden_redirect**(*context*, *request*)

Forbidden redirect view. Redirects to the login form for anonymous users or to the forbidden view for authenticated users.

Result Redirect to one of the above.

Return type pyramid.httpexceptions.HTTPFound

kotti.views.login.**forbidden_view**(*request*)
Forbidden view. Raises 403 for requests not originating from a web browser like device.

Result 403

Return type pyramid.httpexceptions.HTTPForbidden

kotti.views.login.**forbidden_view_html**(*request*)
Forbidden view for browsers.

Result empty dictionary passed to the template for rendering

Return type dict

kotti.views.login.**includeme**(*config*)
Pyramid includeme hook.

Parameters **config** (pyramid.config.Configurator) – app config

kotti.views.site_setup

kotti.views.slots

This module allows add-ons to assign views to slots defined in the overall page. In other systems, these are called portlets or viewlets.

A simple example that'll include the output of the 'hello_world' view in in the left column of every page:

```
from kotti.views.slots import assign_slot
assign_slot('hello_world', 'left')
```

It is also possible to pass parameters to the view:

```
assign_slot('last_tweets', 'right', params=dict(user='foo'))
```

In the view you can get the slot in that the view is rendered from the request:

```
@view_config(name='last_tweets')
def view(request, context):
    slot = request.kotti_slot
    # ...
```

If no view can be found for the given request and slot, the slot remains empty. If you want to force your slot not to be rendered, raise pyramid.exceptions.PredicateMismatch inside your view:

```
from pyramid.exceptions import PredicateMismatch

@view_config(name='last_tweets')
def view(request, context):
    if some_condition:
        raise PredicateMismatch()
    return {...}
```

Usually you'll want to call *kotti.views.slots.assign_slot()* inside an includeme function and not on a module level, to allow users of your package to include your slot assignments through the pyramid.includes configuration setting.

kotti.views.slots.**assign_slot**(*view_name*, *slot*, *params=None*)
Assign view to slot.

Parameters

- **view_name** (*str*) – Name of the view to assign.
- **slot** (*str*) – Name of the slot to assign to. Possible values are: left, right, abovecontent, belowcontent, inhead, beforebodyend, edit_inhead
- **params** (*dict*) – Optionally allows to pass POST parameters to the view.

kotti.views.users

User management screens

kotti.views.users.**name_pattern_validator**(*node*, *value*)

```
>>> name_pattern_validator(None, u'bob')
>>> name_pattern_validator(None, u'b ob')
Traceback (most recent call last):
Invalid: <unprintable Invalid object>
>>> name_pattern_validator(None, u'b:ob')
Traceback (most recent call last):
Invalid: <unprintable Invalid object>
```

kotti.views.users.**includeme**(*config*)

Pyramid includeme hook.

Parameters **config** (pyramid.config.Configurator) – app config

kotti.views.util

class kotti.views.util.**TemplateAPI**(*context*, *request*, *bare=None*, ***kwargs*)

Bases: object

This implements the api object that's passed to all templates.

Use dict-access as a shortcut to retrieve template macros from templates.

static is_location(*context*)

Does *context* implement pyramid.interfaces.ILocation?

Parameters **context** (*kotti.interfaces.INode*) – The context.

Return type bool

Returns True if Is the context object implements pyramid.interfaces.ILocation.

site_title

The site title.

Result Value of the kotti.site_title setting (if specified) or the root item's title attribute.

Return type unicode

page_title

Title for the current page as used in the <head> section of the default master.pt template.

Result '[Human readable view title]''context.title'' - *site_title()*''

Return type unicode

url (*context=None*, **elements*, ***kwargs*)
URL construction helper. Just a convenience wrapper for `pyramid.request.resource_url()` with the same signature. If `context` is `None` the current context is passed to `resource_url`.

root
The site root.

Result The root object of the site.

Return type *kotti.resources.Node*

navigation_root
The root node for the navigation.

Result Nearest node in the *lineage()* that provides *kotti.interfaces.INavigationRoot* or *root()* if no node provides that interface.

Return type *kotti.resources.Node*

lineage
Lineage from current context to the root node.

Result List of nodes.

Return type list of *kotti.resources.Node*

breadcrumbs
List of nodes from the *navigation_root()* to the context.

Result List of nodes.

Return type list of *kotti.resources.Node*

has_permission (*permission*, *context=None*)
Convenience wrapper for `pyramid.security.has_permission()` with the same signature. If `context` is `None` the current context is passed to `has_permission`.

static inside (*resource1*, *resource2*)
Is `resource1` 'inside' `resource2`? Return `True` if so, else `False`.

`resource1` is 'inside' `resource2` if `resource2` is a lineage ancestor of `resource1`. It is a lineage ancestor if its parent (or one of its parent's parents, etc.) is an ancestor.

static sanitize (*html*, *sanitizer='default'*)
Convenience wrapper for *kotti.sanitizers.sanitize()*.

Parameters

- **html** (*unicode*) – HTML to be sanitized
- **sanitizer** (*str*) – name of the sanitizer to use.

Result sanitized HTML

Return type unicode

kotti.views.view

kotti.views.view.**view_content_default** (*context*, *request*)
This view is always registered as the default view for any Content.

Its job is to delegate to a view of which the name may be defined per instance. If a instance level view is not defined for 'context' (in 'context.defaultview'), we will fall back to a view with the name 'view'.

kotti.views.view.**includeme**(*config*)
Pyramid includeme hook.

Parameters **config** (pyramid.config.Configurator) – app config

3.1.18 kotti.workflow

CHAPTER 4

Getting Help / Contributing

4.1 Getting Help

4.2 Contributing

The Kotti project can use your help in developing the software, requesting features, reporting bugs, writing developer and end-user documentation – the usual assortment for an open source project.

Please devote some of your time to the project.

4.2.1 Contributing to the Code Base

To contribute to Kotti itself, and to test and run against the master branch (the current development code base), first create an account on GitHub if you don't have one. Fork Kotti to your github account, and follow the usual steps to get a local clone, with `origin` as your fork, and with `upstream` as the Kotti/Kotti repo. Then, you will be able to make branches for contributing, etc. Please read the docs on GitHub if you are new to development, but the steps, after you have your own fork, would be something like this:

```
git clone https://github.com/your_github/Kotti.git

cd Kotti

git remote add upstream git://github.com/Kotti/Kotti.git
```

Now you should be set up to make branches for this and that, doing a pull request from a branch, and the usual git procedures. You may wish to read the GitHub fork-a-repo help.

To run and develop within your clone, do these steps:

```
virtualenv . --no-site-packages

bin/python setup.py develop
```

This will create a new virtualenv "in place" and do the python develop steps to use the Kotti code in the repo.

Run `bin/pip install kotti_someaddon`, and add a kotti_someaddon entry to `app.ini`, as you would do normally, to use add-ons.

You may wish to learn about the virtualenvwrapper system if you have several add-ons you develop or contribute to. For example, you could have a development area devoted to Kotti work, ~/kotti, and in there you could have clones of repos for various add-ons. And for each, or in some combination, you would use virtualenvwrapper to create virtualenvs for working with individual add-ons or Kotti-based projects. `virtualenvwrapper` will set these virtualenvs up, by default, in a directory within your home directory. With this setup, you can do `workon kotti_this` and `workon kotti_that` to switch between different virtualenvs. This is handy for maintaining different sets of dependencies and customizations, and for staying organized.

4.2.2 Contributing to Developer Docs

Kotti uses the Sphinx tool, using reStructuredText to write documents, stored in docs/ in a directory structure with .rst files. Use the normal git procedures for first making a branch, e.g., `navigation_docs`, then after making changes, commit, push to this branch on your fork, and do a pull request from there, just as you would for contributing to the code base.

In your Kotti clone you can install the requirements for building and viewing the documents locally:

```
python setup.py docs

cd docs/

make html
```

Then you can check the .html files in the _build/ directory locally, before you do an actual pull request.

The rendered docs are built and hosted on readthedocs.org.

4.2.3 Contributing to User Docs

The Kotti User Manual also uses Sphinx and reStructuredText, but there is a bit more to the procedure, because several additional tools are used. Selenium is used for making screen captures, and thereby helps to actually test Kotti in the process. blockdiag is used to make flow charts and diagrams interjected into the docs.

Please follow the readme instructions in the Kotti User Manual repo to get set up for contributing to the user manual. Of course, you can do pull requests that change only the text, but please get set up for working with graphics also, because this is a way to do the important task of keeping Kotti user docs up-to-date, guaranteed to have graphics in sync with the latest Kotti version.

The rendered docs are built and hosted on readthedocs.org.

CHAPTER 5

Future and Past

5.1 Change History

5.1.1 1.3.1.dev0 - unreleased

- When rendering slot views, use `request.blank()` to create the request. This is the proper behaviour, in tune with customizing `kotti.request_factory`. Also added `blank()` method to `kotti.testing.DummyRequest`.
- When authenticated, show workflow state in the edit bar. Before it was shown only if the 'edit' permission was available.
- Optimize the File edit form: don't load initial file data to session data and don't rewrite the file data after saving the form if that data has not been changed through the edit form.
- Bugfix: when showing addable content in the menu, check if the factory has a defined add_view. This avoids a hard crash with, for example, a content type derived from Content that has no add_view defined.
- Added `nav-bar` slot to `edit/master.pt`, `edit-bar` and `nav-bar` slots to `view/master.pt`
- Bugfix: Simplify 404 page, no longer crash when authenticated
- Change: simplify *kotti.util.LinkBase.selected()*: use request.view_name instead of deriving the view name from request.url. Also, consider the View editor bar entry as selected even when the url doesn't end with a slash '/'
- Feature: add Czech translation.
- Switch from `oursql``to ``mysqlclient` in tests.
- Setup tests on TravisCI with `pip install -e .[testing]`, making `requirements.txt` obsolete.
- Adjust CLI command tests for new versions of `Pyramid`/`plaster`.
- remove pytest-warnings from test dependencies (already integrated in modern pytest versions)

5.1.2 1.3.0 - 2016-10-10

Breaking Changes

- Upgrade to `repoze.workflow==1.0b`. **If your application has a custom ‘‘workflow.zcml‘‘, it needs a little modification: ‘‘state‘‘ and ‘‘transition‘‘ titles are no longer ‘‘key‘‘ nodes, but attributes on the respective ‘‘state‘‘ or ‘‘transition‘‘ nodes. See Kotti’s ‘‘workflow.zcml‘‘ for an example.**

Features and Fixes

- Add a fallback in `contents.pt` when `creation_date` or `modification_date` is `None`.
- Transform workflow state title to TranslationStrings without eval and deprecate it.
- Replace some Python 2 only code with equivalents that also support Python 3.
- Use generic SQLAlchemy type Text as base type for JsonType. This allows SQLAlchemy to map Text type to the most suitable type available on given database system. Previously used TEXT type is not available in Oracle database. In case of existing installation of Kotti with database system, for which SQLAlchemy maps generic Text type to type different than TEXT it’s necessary to either convert existing columns “nodes._acl” and “nodes.annotations” to that type or configure SQLAlchemy to map generic Text type to existing type of these two columns. For example of how to do this please see http://stackoverflow.com/a/36506666/95735. For all database systems for which SQLAlchemy provides dialects except Oracle (Firebird, Microsoft SQL Server, MySQL, Postgres, SQLite, Sybase) there’s no need to do anything.
- We use PEP 440 normalized form for the project’s version thus current “1.3.0-alpha.5-dev” became “1.3.0a5.dev0”.
- Upgrade tests to `zope.testbrowser>=5.0.0`. This removes the `mechanize` and `wsgi_intercept` dependencies and thus the last blocker for Python 3 compatibility.
- Move `pytest` config from `setup.cfg` to new `pytest.ini`. This prevents a deprecation warning with `pytest>=3.0`.
- Rename `kotti.testing.TestingRootFactory` to `kotti.testing.RootFactory` to prevent another deprecation warning with `pytest>=3.0`.

5.1.3 1.3.0-alpha.4 - 2015-01-15

This is a alpha release. Blindly upgrading your production environments will make the universe collapse!

- Add a `kotti.depot_replace_wsgi_file_wrapper` option to replace the WSGI file wrapper with `pyramid.response.FileIter` for problematic environments.

5.1.4 1.3.0-alpha.3 - 2016-01-11

This is a alpha release. Blindly upgrading your production environments will make the universe collapse!

- Bugfix: don’t try to get api.root via the lineage if not in a location aware context (for example 404 view). Return the site root instead.

5.1.5 1.3.0-alpha.2 - 2016-01-05

This is a alpha release. Blindly upgrading your production environments will make the universe collapse!

- Add a custom traverser, which gets all nodes in a single DB query. For deeply nested trees this results in drastic performance improvements. See https://kotti.readthedocs.io/en/master/api/kotti.traversal.html for details.
- Bugfix: copy and paste of file nodes wouldn't create a new depot file, but instead lead to multiple references to a single file which would cause undesired results when one of them was deleted later.
- Bugfix: local 'role:owner' was not set when a new node was created by copy and paste.
- Bugfix: kotti.events._update_children_paths could fail under unclear conditions (at least under Python 2.6 with SQLite).
- Get rid of more browser doctests (converted to webtest).

5.1.6 1.3.0-alpha.1 - 2015-12-22

This is a alpha release. Blindly upgrading your production environments will make the universe collapse!

- Completely revised `Depot` integration. See https://kotti.readthedocs.io/en/latest/developing/advanced/blobs.html for details.
- Make `kotti.resources.SaveDataMixin` more versatile in that it now supports a `data_filters` attribute (or even a completely overridden `data` attribute) on subclasses. For an example for what this is useful, see the new `kotti_image` package's readme and the Depot documentation (https://depot.readthedocs.io/en/latest/database.html#custom-behaviour-in-attachments).

  These changes require a database migration.

  A migration script is included, which can be executed by running `kotti-migrate <your.ini> upgrade_all`. However, this script will fail if you subclassed from `kotti.resources.Image` in your application. It also doesn't cover custom classes inherited from `kotti.resources.File` (other than Kotti's `Image`). Migration of those can be performed easily, by copying the code from the included migration step to your package's migration environment and adjust it to your needs.
- Move all image related code to the new `kotti_image` add on package. All classes and functions are imported into their former place, so that code that imports from there will still be working.
- Fix broken upload type selector.
- Create RFC6266 compliant content disposition headers for non-ASCII filenames.
- Add request.uploaded_file_response method.

5.1.7 1.2.4 - 2015-11-26

- Fix broken packaging of 1.2.3. Sorry for the inconvenience!

5.1.8 1.2.3 - 2015-11-26

- Add Kotti logo and icon to static assets.
- Use Kotti logo as favicon.
- Move favicon definition to separate template to make it easily overridable.
- Fix permission check in `kotti.views.util.nodes_tree`.

5.1.9 1.2.2 - 2015-10-28

- Add simple, default `not found view`.
- In `workflow-dropdown` replace hard-coded permission check with individual permission checks for each existing transition.
- Upgrade requirements.

5.1.10 1.2.1 - 2015-10-07

- Outfactor the code that runs after successful authentication into a configurable `kotti.login_success_callback` function.
- Outfactor the code that runs after a valid password reset request into a configurable `kotti.password_reset_callback` function.
- Support principal search on non string attributes.
- Support principal searches matching *all* arguments (i.e. using the `and` operator, `or` is still the default).
- Support optional –rev with kotti-migrate upgrade.

5.1.11 1.2.0 - 2015-09-27

- **Greatly** reduce the number of queries that are sent to the DB: - Add caching for the root node. - Use eager / joined loading for local_groups. - Don't query principals for roles
- Add "missing" foreign key indices (with corresponding migration step).
- Add a `kotti.modification_date_excludes` configuration option. It takes a list of attributes in dotted name notation that should not trigger an update of `modification_date` on change. Defaults to `kotti.resources.Node.position`.
- Don't try to set a caching header from the NewRequest handler when Pyramid's tweens didn't follow the usual chain of calls. This fixes compatibility with `bowerstatic`.
- Don't assume `renderer_name` exists in a rendering event (ex. BeforeRender). The official docstring of `pyramid.interfaces.IRenderer` is a bit ambigous in regards to what the `system` parameter should include when a renderer gets called. This fixes compatibility with `pyramid_layout`.
- Add a `kotti.modification_date_excludes` configuration option. It takes a list of attributes in dotted name notation that should not trigger an update of `modification_date` on change. Defaults to `kotti.resources.Node.position`.

5.1.12 1.1.5 - 2015-09-04

- Fix migration error on MySQL.
- Only wrap methods that do exist on the wrapped type (in `kotti.sqla.MutationList` / `kotti.sqla.MutationDict`). This fixes an error that occurs when MutationLists are exposed to the UI via `colander.SequenceSchema`.
- Upgrade requirements to latest versions (`filedepot`, `waitress`).

5.1.13 1.1.4 - 2015-06-27

- Add compatibility with SQLAlchemy 1.0. Also require SQLAlchemy 1.0.6 now.
- Ignore HTTPForbidden exceptions during slot rendering

5.1.14 1.1.3 - 2015-06-17

- Fix a bug in kotti-migrate that prevented initial migration steps from being run successfully.
- Require kotti_tinymce 0.5.3.

5.1.15 1.1.2 - 2015-06-12

- Enlarge column sizes for `name`, `path` and `title` (see #427). Upgrading from any version older than 1.1.2 requires you to run a migration script on your database. To run the migration, call:

```
$ bin/kotti-migrate <myconfig.ini> upgrade
```

- Add length validator for `title` (fix partially #404). See #428
- Remove 40 chars max length constraint for the html segment `name` (`Kotti.util.title_to_name`). See #428
- Update italian translation
- Update documentation
- Add an `add_permission` attribute to `kotti.resources.TypeInfo` with a default value of `add`. See #436
- Add a "cancel" button to the delete node view.

5.1.16 1.1.1 2015-05-11

- Update scaffold's README file. See #417.
- Fix broken multifile upload. See #425.

5.1.17 1.1.0 2015-04-16

- Separate the default actions to a `kotti.resources.default_actions` variable, to allows easier customization of default actions of all content types. This is a `LinkParent`, you can append new `kotti.util.Link` objects to its `children`.
- Add `target` option to `kotti.util.Link`. See #405.
- Add sanitizers. See `docs/development/advanced/sanitizers` and `kotti.sanitizers` for details. This fixes #296.
- Added new document on how to customize the edit interface. See `docs/development/advanced/add-to-edit-interface`.
- Make it easier to customize default actions by separating them to a new `kotti.resources.default_actions` variable. Before, to customize them, you'd have to change `Content.type_info.edit_links[3].children`, now you can mutate `default_actions` directly. See `docs/development/advanced/add-to-edit-interface` for details.

- Upgrade `WebOb`, `html2text`, `pyramid` and `xlwt` to their latest stable versions.

5.1.18 1.1.0-alpha.1 - 2015-03-19

- Allow moving `File` and `Image` blob data from the database to configurable storages. To achieve this we use filedepot, a third-party library with several plugin storages already built in. See docs/developing/advanced/blobs.rst for details on what this brings. Upgrading from any version older then 1.1.0 requires you to run a migration script on your database. To run the migration, call:

```
$ bin/kotti-migrate <myconfig.ini> upgrade
```

  Please note that, before running the migration, you should take the time to read the documentation and configure your desired storage scheme.

- Allow storing blob data in the database using `DBStoredFile` and `DBFileStorage`, a database centered storage plugin for `filedepot`. This storage is the default storage for blob data, unless configured otherwise.
- Added a script to migrate blob data between depot storages. See docs/developing/advanced/blobs.rst for details on how to use it.
- Simplify serving blob data by using `kotti.views.file.UploadedFileResponse`, which also streams data. Please note that the default `DBStoredFile` still needs to load its entire data in memory, to benefit from this feature you should configure another default depot storage.
- Added three new test fixtures: `mock_filedepot`, to be used in simple unit tests with no dependency on a database session, `filedepot`, which integrates with the `dbsession` fixture and `no_filedepot`, a fixture that can be used in developing tests for new file depot plugins - by preserving the depot configuration before and after running the test. NOTE: in order to test edit views with uploaded data in the request, you need to mixin the `filedepot` fixture.
- Initialize pyramid.paster.logging for custom commands defined via `kotti.util.command`, to allow log message output for kotti sessions started via custom commands.
- Remove unused `kotti.js`.
- Remove deprecated `kotti.views.slots.local_navigation` and `kotti.views.slots.includeme_local_navigation`. Use `kotti.views.navigation.local_navigation` and `kotti.views.navigation.includeme_local_navigation` instead.
- Upgrade `plone.scale` and `SQLAlchemy` to their latest stable versions.
- Change `height` property on `body`'s widget (`RichTextField`) for improved usability. See #403.

5.1.19 1.0.0 - 2015-01-20

- No changes.

5.1.20 1.0.0-alpha.4 - 2015-01-29

- Added experimental Docker support. See #374.
- Allow restricting *add views* to specific contexts. This allows third party developers to register new content types that are addable in specific type of contexts, by specifying `context=SomeContentType` in their *add view* registration and having `type_info.addable=['SomeContentType']` in the type info.
- For documents with duplicate titles that end in a number, append a counter instead of incrementing their number. Fixes #245

- Update all requirements (except alembic) to their latest respective versions.

5.1.21 1.0.0-alpha.3 - 2015-01-13

- Explicitly implement `pyramid.interfaces.IRequest` for `kotti.request.Request`. This allows add-on packages to use `config.add_request_method` (with reify) and `config.add_request_property` without breaking the interfaces provided by the request. Fixes #369

5.1.22 1.0.0-alpha.2 - 2015-01-01

- Require `kotti_tinymce==0.5.1`. This fixes #365.

5.1.23 1.0.0-alpha - 2014-12-20

- Add a new scaffold based on Pyramid's `pcreate`. To run the tests for the scaffold, you must invoke `py.test` with the `--runslow` option. This is enabled by default on Travis.
- `kotti._resolve_dotted` now return a resolved copy of the settings (instead of in place resolving as before).
- Factor out DBMS specific patches and make them available to the test fixtures.
- Add new fixtures that can also be used in add on tests:
  - `custom_settings` does nothing and is meant to be overridden in add on test suites. It allows injection of arbitrary key / values into the settings dict used in tests.
  - `unresolved_settings` is guaranteed to only contain unresolved string values (or lists therof).
  - `settings` is now guaranteed to be fully resolved.
  - `webtest` returns a `webtest.TestApp` instance with support for the `@user` marker. This should be used instead of browser doctests for functional tests.
- Use RTD theme for documentation.
- Use latest versions of all requirements. The only upgrade with notable differences is `lingua` (from 1.4 to 3.6.1). This completely changes lingua's API. See docs/developing/basic/translations.rst for details on the greatly simplified new usage.
- Remove code (incl. tests) that has been marked as deprecated since (at least) Kotti 0.8.
- Revise UI to make better use of Bootstrap 3.
- Allow parameters for move-child-position views to either be in `request.POST` or `request.json_body`.
- Don't use Pyramid code that is marked as deprecated:
  - replace `pyramid.security.authenticated_userid` with `request.authenticated_userid`.
- Deprecate `kotti.security.has_permission` to be consistent with the corresponding deprecation in Pyramid 1.5. You should now use `request.has_permission` instead.
- Make all values in `Node.path` end in `/`. This makes it consistent over all nodes (*including* root) and correspond to the values of `request.resource_url`. As a side effect querying becomes easier. However, this might need adjustments in your code if you were expecting the old path values before. A migration step for DB upgrades is included.

5.1.24 0.10b1 - 2014-07-11

- Add a `__json__` method to MutationList and MutationDict.

  This is to allow Pyramid's serializer to just work.

5.1.25 0.10a4 - 2014-06-19

- Upgrade Pyramid to version 1.5.1.

5.1.26 0.10a3 - 2014-06-11

- Upgrade SQLAlchemy and alembic dependencies from 0.8.2 and 0.5.0 to 0.9.4 and 0.6.5 respectively.
- Do not flush within `Node.path` event handlers. We would otherwise trigger object handlers with funny object states.
- Fix bug with `Node.path` where we attach a Node instance to a parent that has been loaded from the database, but its parents have not been loaded yet.
- Fix deprecation warnings with regard to Pyramid's `custom_view_predicates` and `set_request_property`. Also deprecate `kotti.views.util.is_root`.

5.1.27 0.10a2 - 2014-06-05

- Add `Node.path` column. This allows queries based on path, so it's much easier just to find all children, grandchildren etc. of a given node:

```
DBSession.query(Node).filter(Node.path.startswith(mynode.path))
```

- Adds `session` attribute to the request attributes to copy to the slot view request.

Migrations

- Upgrading from 0.9.2 to 0.10 requires you to run a migration script on your database. To run the migration, call:

```
$ bin/kotti-migrate <myconfig.ini> upgrade
```

  Make sure you **backup** your database before running the migration!

5.1.28 0.10a1 - 2014-05-19

- Kotti is now based on Bootstrap 3 (and therefore Deform 2).

  THIS IS A BACKWARD INCOMPATIBLE CHANGE W.R.T. MOST TEMPLATES, INCLUDING FORM TEMPLATES! IF YOUR PROJECT EITHER HAS TEMPLATE CUSTOMIZATIONS OR DEPENDS ON ADD-ONS THINGS WILL LOOK BROKEN!

  If you **only** use Kotti's default UI, chances are good that your application will continue to work well unchanged. Kotti's API is mostly unchanged and fully backward compatible though.

- Rework implementation of 'kotti.util.Link' ('ViewLink') to be more flexible.

  There's now proper support for nesting 'edit_links', so that the special 'action_links' list is no longer necessary. Links now also make better use of templates for rendering, and are probably easier to customize overall.

- Added compatiblity for and now require Pyramid>=1.5. #273
- In tests, turned *settings* and *setup_app* into fixtures to ease overriding.
- Add `kotti_context_url` JS global variable. For more details on why this is needed see:
  - https://github.com/Kotti/kotti_tinymce/issues/19
  - https://github.com/Kotti/Kotti/issues/219
  - https://github.com/Kotti/kotti_newsitem/issues/2
  - https://github.com/Kotti/kotti_calendar/issues/4
- Adds `delete` permission needed for 'delete' and 'delete_nodes' views. The default workflow was updated in consequence. It allows to elaborate more fine grained workflows : for instance, create a role which can edit a content but not delete it.

  To make existent Kotti's instances using default workflow compatibles and avoid users that have 'editor' role (and so far, whom have the possibility to edit and delete the content) to not be able to delete contents, it's needed to reset workflow with "kotti-reset-workflow <application ini file>" command.
- Fix #308: Unique name constraint issue during paste of a cut node.

5.1.29 0.9.2 - 2013-10-15

- Fix #268: Convert None to colander.null in get_appstruct so that serialization doesn't fail (needed due to recent changes in colander).

5.1.30 0.9.1 - 2013-09-25

- Allow user admins to modify user passwords.
- Require newer kotti_tinymce (source code editing was broken in 0.4).

5.1.31 0.9 - 2013-09-17

- Add multi file content upload. You can now select several files from your local storage that you want to upload and chose what content nodes shall be created in your Kotti site. Currently files with MIME types of `image/*` can be uploaded and be created as either `Image` or `File` nodes, all other MIME types will be created as `File`. In future releases (or add-on products) this can be extended with additional converters allowing for example to upload HTML files and create `Document` nodes with the content of the `title` tag becoming the node's title, the content of the `body` tag becoming the node's body and so on.
- Fix #253: Many translations weren't included in the last release.

  '–use-fuzzy' translations when running 'compile_catalog' adds back translations that were recently marked as fuzzy. (All translations that were marked as fuzzy in German were still accurate.)
- Fix #252: Wrap templates where extract_messages failed with <tal:block>
- Fix #249: TinyMCE translations work again.

5.1.32 0.9b2 - 2013-08-20

- Fix #251: Broken comparison of NestedMutationDict and NestedMutationList.
- Update kotti_tinymce to version 4.0.2.

- Fix bug in *kotti.views.content.FileEditForm* to preserve file content while editing it.

5.1.33 0.9b1 - 2013-06-26

- Add `kotti.util.ViewLink.visible` method for better control over whether a view link should be visible or not. This allows us to move formerly hardcoded action links defined in `kotti.views.edit.actions` into `TypeInfo.action_links` and thus make them configurable either globally or per content type.
- `kotti.security.view_permitted` will now check for `pyramid.security.view_execution_permitted` with a request method set to 'GET' by default. It used to check for a view that matches the current request's method.

  This fixes an issue where `kotti.util.ViewLink.permitted` would by mistake check for a 'POST' view when the current request was 'POST'.
- Add `INavigationRoot` interface and `TemplateAPI.navigation_root` property. The latter returns the first content node in the lineage that implements `INavigationRoot` or the root node if `INavigationRoot` is not implemented by any node in the lineage. Make the `nav.pt` template use `api.navigation_root` instead of `api.root`. This allows third party add-ons to define content types that can reside somewhere in the content tree while still being the root for the navigation.
- Move navigation related view code to new module `kotti.views.navigation`. Deprecate imports from the old locations.
- Remove *some* code that has been deprecated in 0.6 or 0.7.
- A view assigned to a slot can access the slot name where its rendered.
- Add missing *transaction.commit()* in *kotti-reset-workflow*.
- Fix bug in *kotti.views.util.render_view* where local roles weren't respected correctly.
- Add helper method *kotti.message.send_email* for sending general emails. These emails must follow a particular structure. Look at *kotti:templates/email-set-password.pt* as an example.

5.1.34 0.9a2 - 2013-05-04

- Fix #222: Use SQLAlchemy's *before_flush* event for object events.

  We were using the wrong events previously. The problem with *before_insert*, *before_update*, and *before_delete* was that event handlers could not reliably call *Session.add*, *Session.delete*, and change mapped relationships. But only SQLAlchemy 0.8 started emitting a warning when that was done.

  Also deprecated *ObjectAfterDelete* because I don't think it's useful.
- Remove the html5shim from the master templates and use the fanstatic package js.html5shiv instead.
- A temporary fix for #187. Basically suppresses DetachedInstanceError.
- Add `kotti.events.subscribe` decorator. See the also updated docs on that topic / module for details.

5.1.35 0.9a1 - 2013-03-12

- Fix ordering on how `include_me` functions are loaded. This puts Kotti's own and Kotti add-on search paths in front of deform_bootstrap's.
- Add image thumbs with preview popovers to @@contents view.

- Add drag'n'drop ordering support to @@contents view.
- Add "toggle all" checkbox to @@contents view.
- Add contents path bar to @@contents view.

5.1.36 0.8 - 2013-03-12

- No changes.

5.1.37 0.8b2 - 2013-02-08

- Fix Kotti's tests to no longer trigger deprecation warnings. Kotti's funcargs need to be better documented still, see #141.
- Add a fanstatic.Group 'tagit' and need() it in the defered widget. This is needed to make the tags widget render correctly with a theme package enabled until the defered widget is replaced by a widget class that declares its requirements in the usual deform style.
- Transform `setup_users`, `setup_user` and `prefs` views into class-based views. Add a little text at subsection `Security` on developer manual mentioning those views.

5.1.38 0.8b1 - 2012-12-30

- No changes

5.1.39 0.8a2 - 2012-12-15

- Remove test related dependencies on requirements.txt. So now we need to run *python setup.py dev* to get testing dependencies.
- Update packages versions on requirements.txt for latest working versions.
- Added a tags display in views for documents, files, folders, and images, where they show up as a horizontal list between description and body.
- Modified general search to include simple tags searching. The default search in Kotti works on a simple search term matching basis. Tags searching is added here in a simple fashion also, such that you can only search for one tag at a time, but partial matches work: searching for 'foo' finds content tagged 'foo bar'. You can also search on single tags by clicking an individual tag in the tags display of an item. More sophisticated tags searching, just as for general search, is left to dedicated add-ons.

5.1.40 0.8a1 - 2012-11-13

- Make language-dependent URL normalization the default. (How to do this used to be a cookbook entry.)
- Cleanup node edit actions and use decorated view classes.
- Add contents view with actions for multiple items.
- Add children_with_permission method to ContainerMixin.
- Add UI for default_view selection.
- Deprecate 'kotti.views.edit.generic_add' and 'generic_edit'. Just use class-based forms instead.

5.1.41 0.7.2 - 2012-10-02

- Improve installation instructions. Now uses tagged requirements.txt file.
- Added event request POST vars to the request for the slot viewlet.
- Added IFile and IImage interfaces to allow for file and image subclasses to reuse the same view (registrations).

5.1.42 0.7.1 - 2012-08-30

- Add deletion of users to the users management.
- Fix tag support for files and images.
- Upgrade to Twitter Bootstrap 2.1
  - remove lots of CSS that is no longer needed
  - fix responsive layout that was broken on some phone size screen resolutions
- Add "Site Setup" submenu / remove @@setup view.

5.1.43 0.7 - 2012-08-16

- Fix critical issue with migrations where version number would not be persisted in the Alembic versions table.

5.1.44 0.7rc1 - 2012-08-14

- No changes.

5.1.45 0.7a6 - 2012-08-07

- Fix a bug with connections in the migration script. This would previously cause Postgres to deadlock when calling `kotti-migrate`.

5.1.46 0.7a5 - 2012-08-07

- Add workflow support based on `repoze.workflow`. A simple workflow is included in `workflow.zcml` and is active by default. Use `kotti.use_workflow = 0` to deactivate. The workflow support adds a drop-down that allows users with `state_change` permission to modify the workflow state.
- Change the default layout

  Kotti's new default look is now even closer to the Bootstrap documentation, with the main nav bar at the very top and the edit bar right below.

  Upgrade note: if you have a customized main_template and want to use the recent changes in that template, you need to swap positions of `nav.pt` and `editor-bar.pt` `api.render_template` calls and remove the `search.pt` call from the main_template (it's now called from within `nav.pt`). Everything else is completely optional.
- Add migrations via Alembic. A new script `kotti-migrate` helps with managing database upgrades of Kotti and Kotti add-ons. Run `kotti-migrate <your.ini> upgrade` to upgrade the Kotti database to the latest version.

  Add-on authors should see the `kotti.migrate` module's docstring for more details.

- Make `Document.body` searchable (and therefore the search feature actually useful for the first time).
- Add a "minify" command to compress CSS and JS resources.

  To use it run:

```
python setup.py dev
python setup.py minify
```

  The `minify` command assumes, that all resources are in `kotti/static/`. `YUI compressor` is used for compression and will be automatically installed when running `python setup.py dev`. However, you still need a JVM on your development machine to be able to use the `minify` command.
- Fix settings: only values for kotti* keys should be converted to unicode strings.
- Fix #89: Validate email address for uniqueness when user changes it.
- Fix #91: Styling of search box.
- Fix #104: Make fanstatic resources completely overridable.
- Enabled deferred loading on File.data column.

Migrations

- Upgrading from 0.6 to 0.7 requires you to run a migration script on your database. To run the migration, call:

```
$ bin/kotti-migrate <myconfig.ini> upgrade
```

  Make sure you **backup** your database before running the migration!
- Upgrading to 0.7 will initialize workfow state and permissions for all your content objects, unless you've overwritten `kotti-use_workflow` to not use a workflow (use `0`) or a custom one.

  It is **important** that sites that have custom permissions, e.g. custom modifications to SITE_ACL, turn off workflow support prior to running the upgrade script.

5.1.47 0.7a4 - 2012-06-25

- Add minified versions JS/CSS files.
- Fix #88: logging in with email.
- Update translations.

5.1.48 0.7a3 - 2012-06-15

- Include `kotti.tinymce` which adds plug-ins for image and file upload and content linking to the TinyMCE rich text editor.
- Slot renderers have been replaced by normal views (or viewlets). `kotti.views.slots.register` has been deprecated in favour of `kotti.views.slots.assign_slot`, which works similarly, but takes a view name of a registered view instead of a function for registration.
- Switch to fanstatic for static resource management.

  Upgrade note: This requires changes to existing **.ini* application configuration files. Concretely, you'll need to add a `filter:fanstatic` section and a `pipeline:main` section and rename an existing `app:main` section to `app:Kotti` or the like. Take a look at Kotti's own `development.ini` for an example.

- Retire the undocumented `kotti.resources.Setting` class and table. `kotti.get_settings` will now return `registry.settings` straight, without looking for persistent overrides in the database.
- Drop support for Pyramid<1.3, since we use `pyramid.response.FileResponse`, and kotti_tinymce uses `pyramid.view.view_defaults`.
- Fix encoding error with non-ascii passwords.

5.1.49 0.7a2 - 2012-06-07

- Do not allow inactive users to reset their password.

5.1.50 0.7a1 - 2012-06-01

Features

- Add a new 'Image' content type and image scaling, originally from the kotti_image_gallery add-on. See `kotti.image_scales.*` settings.
- Add search and related setting `kotti.search_content`.
- Add subscriber to set cache headers based on caching rules. See also related setting `kotti.caching_policy_chooser`.
- Remove TinyMCE from the core.
- Move email templates into page templates in `kotti:templates/email-set-password.pt` and `kotti:templates/email-reset-password.pt`. This is to make them easier to translate and customize. This deprecates `kotti.message.send_set_password`.
- Add a 'edit_inhead' slot for stuff that goes into the edit interface's head. 'inhead' is no longer be used in 'edit/master.pt'.
- For more details, see also http://danielnouri.org/notes/2012/05/28/kotti-werkpalast-sprint-wrap-up/

Bugs

- Fix bug with group edit views. See https://github.com/Pylons/Kotti/pull/61
- Fix bug where `user.last_login_date` was not set during automic login after the set password screen.

5.1.51 0.6.3 - 2012-05-08

- Add tag support. All content objects now have tags. They can be added in the UI using the "jQuery UI Tag-it!" widget. See https://github.com/Pylons/Kotti/pull/55 .
- Fix a bug with file download performance.

5.1.52 0.6.2 - 2012-04-21

- Links in Navigation view lead to node view. Added edit links to view the node's edit form.
- Hitting 'Cancel' now returns to the context node for add/edit views

5.1.53 0.6.1 - 2012-03-30

- Added button to show/hide nodes from navigation in the order screen.
- The 'rename' action now strips slashes out of names. Fixes #53.
- Add Dutch translation.
- Allow translation of TinyMCE's UI (starting with deform 0.9.5)
- Separated out testing dependencies. Run `bin/python setup.py dev` to install Kotti with extra dependencies for testing.
- Deprecate 'kotti.includes' setting. Use the standard 'pyramid.includes' instead.
- Setting 'Node.__acl__' to the empty list will now persist the empty list instead of setting 'None'.
- Let 'pyramid_deform' take care of configuring deform with translation dirs and search paths.

5.1.54 0.6.0 - 2012-03-22

- Add Japanese translation.
- Enforce lowercase user names and email with registration and login.
- Moved SQLAlchemy related stuff from kotti.util into kotti.sqla.
- You can also append to 'Node.__acl__' now in addition to setting the attribute.

5.1.55 0.6.0b3 - 2012-03-17

- Have the automatic `__tablename__` and `polymorphic_identity` for CamelCase class names use underscores, so a class 'MyFancyDocument' gets a table name of 'my_fancy_documents' and a type of 'my_fancy_document'.

5.1.56 0.6.0b2 - 2012-03-16

- Make the 'item_type' attribute of AddForm optional. Fixes #41.
- `kotti.util.title_to_name` will now return a name with a maximum length of 40. Fixes #31.

5.1.57 0.6.0b1 - 2012-03-15

- Use declarative style instead of class mapper for SQLAlchemy resources.

  Unfortunately, this change is backwards incompatible with existing content types (not with existing databases however). Updating your types to use Declarative is simple. See kotti_calendar for an example: https://github.com/dnouri/kotti_calendar/commit/509d46bd596ff338e0a88f481339882de72e49e0#diff-1

5.1.58 0.5.2 - 2012-03-10

- A new 'Actions' menu makes copy, paste, delete and rename of items more accessible.
- Add German translation.
- Populators no longer need to call `transaction.commit()` themselves.

5.1.59 0.5.1 - 2012-02-27

- Internationalize user interface. Add Portuguese as the first translation.
- A new 'Add' menu in the editor toolbar allows for a more intuitive adding of items in the CMS.
- Refine `Node.copy`. No longer copy over local roles per default.

5.1.60 0.5.0 - 2012-02-15

- Move Kotti's default user interface to use Twitter Bootstrap 2.
- Add a new 'File' content type.
- Add CSRF protection to some forms.
- Remove Kotti's `FormController` in favor of using `pyramid_deform`.
- Use plone.i18n to normalize titles to URL parts.
- Add a separate navigation screen that replaces the former intelligent breadcrumbs menu.
- Use `pyramid_beaker` as the default session factory.
- Make `kotti.messages.send_set_password` a bit more flexible.

5.1.61 0.4.5 - 2012-01-19

- Add 'kotti.security.has_permission' which may be used instead of 'pyramid.security.has_permission'.

  The difference is that Kotti's version will set the "authorization context" to be the context that you pass to 'has_permission'. The effect is that 'list_groups' will return a more correct list of local roles, i.e. the groups in the given context instead of 'request.context'.
- Add a template ('forbidden.pt') for when user is logged in but still getting HTTPForbidden.

5.1.62 0.4.4 - 2012-01-05

- The "Forbidden View" will no longer redirect clients that don't accept 'text/html' to the login form.
- Fix bug with 'kotti.site_title' setting.

5.1.63 0.4.3 - 2011-12-22

- Add 'kotti.root_factory' setting which allows the override Kotti's default Pyramid *root factory*. Also, make master templates more robust so that a minimal root with '__parent__' and '__name__' can be rendered.
- The 'kotti.tests' was factored out. Tests should now import from 'kotti.testing'.

5.1.64 0.4.2 - 2011-12-20

- More convenient overrides for add-on packages by better use of 'config.commit()'.

5.1.65 0.4.1 - 2011-12-20

- Modularize Kotti's Paste App Factory 'kotti.main'.
- Allow explicit setting of tables that Kotti creates ('kotti.use_tables').

5.1.66 0.4.0 - 2011-12-14

- Remove configuration variables 'kotti.templates.*' in favour of 'kotti.asset_overrides', which uses Pyramid asset specs and their overrides.
- Remove 'TemplateAPI.__getitem__' and instead add 'TemplateAPI.macro' which has a similar but less 'special' API.
- Factor snippets in 'kotti/templates/snippets.pt' out into their own templates. Use 'api.render_template' to render them instead of macros.

5.1.67 0.3.1 - 2011-12-09

- Add 'keys' method to mutation dicts (see 0.3.0).

5.1.68 0.3.0 - 2011-11-30

- Replace *Node.__annotations__* in favor of an extended *Node.annotations*.

  Node.annotations will attempt to not only recognize changes to subobjects of type dict, it will also handle list objects transparently. That is, changing arbitrary JSON structures should just work with regard to calling *node.annotations.changed()* when the structure was changed.

5.1.69 0.2.10 - 2011-11-22

- 'api.format_datetime' now also accepts a timestamp in addition to datetime.

5.1.70 0.2.9 - 2011-11-21

- Remove MANIFEST.in in favour of using 'setuptools-git'.

5.1.71 0.2.8 - 2011-11-21

- Remove 'PasteScript' dependency since that would result in spurious errors when installing Kotti. See http://jenkins.danielnouri.org/job/Kotti/42/TOXENV=py27/console

5.1.72 0.2.7 - 2011-11-20

- Add 'PasteScript' dependency.
- Fix #11 where 'python setup.py test' would look into a hard-coded 'bin' directory.
- Structural analysis documentation. (Unfinished; in 'analysis' directory during development. Will be moved to main docs when finished.)

5.1.73 0.2.6 - 2011-11-17

- Add *Node.__annotations__* convenience attribute.

  Node.__annotations__ will wrap the annotations dict in such a way that both item and attribute access are possible. It'll also record changes to dicts inside dicts and mark the parent *__annotations__* attribute as dirty.

- Add a welcome page.
- Delete the demo added in version 0.2.4.

5.1.74 0.2.5 - 2011-11-14

- Add 'TemplateAPI.render_template'; allow templates to be rendered conveniently from templates.

5.1.75 0.2.4 - 2011-11-13

- Adjust for Pyramid 1.2: INI file, pyramid_tm, Wsgiref server, pcreate and pserve. (MO)
- Add Kotti Demo source and documentation.

5.1.76 0.2.3 - 2011-10-28

- `Node.__getitem__` will now also accept a tuple as key.

  `folder['1', '2']` is the same as `folder['1']['2']`, just more efficient.

- Added a new cache decorator based on `repoze.lru`.

5.1.77 0.2.2 - 2011-10-10

- Change the function signature of `kotti.authn_policy_factory`, `kotti.authz_policy_factory` and `kotti.session_factory` to include all settings from the configuration file.

5.1.78 0.2.1 - 2011-09-29

- Minor changes to events setup code to ease usage in tests.

5.1.79 0.2 - 2011-09-16

- No changes.

5.1.80 0.2a2 - 2011-09-05

- Fix templates to be compatible with Chameleon 2. Also, require Chameleon>=2.
- Require pyramid>=1.2. Also, enable pyramid_debugtoolbar for `development.ini` profile.

5.1.81 0.2a1 - 2011-08-29

- Improve database schema for `Nodes`. Split `Node` class into `Node` and `Content`.

  This change is backward incompatible in that existing content types in your code will need to subclass `Content` instead of `Node`. The example in the docs has been updated. Also, the underlying database schema has changed.

- Improve user database hashing and local roles storage.
- Compatibility fix for Pyramid 1.2.

Python Module Index

k

Index

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X