
TurboGears Documentation

Release 2.3.5

The TurboGears Doc Team

April 28, 2015

1	Tutorials	3
1.1	Hello TurboGears	3
1.2	Full Featured TurboGears: A Wiki in 20 Minutes	6
1.3	Rapid Prototyping: A Wiki using the TurboGears Admin	20
2	Basic Documentation	33
2.1	Writing Controllers	33
2.2	Templating	38
2.3	Genshi XML Template Language	39
2.4	Scaffolding	48
2.5	TurboGears Validation	48
2.6	Displaying Flash/Notice Messages	53
2.7	Authorization in TurboGears	55
2.8	Web Session Usage	60
2.9	Caching	62
2.10	Handling Internationalization And Localization	67
2.11	Writing TurboGears Extensions	71
2.12	Pluggable Applications with TurboGears	73
2.13	Pagination in TurboGears	74
2.14	Using MongoDB	76
3	Advanced Documentation	79
3.1	Installing TurboGears2	79
3.2	The GearBox Toolkit	81
3.3	RESTful Web Applications with TurboGears	83
3.4	ObjectDispatch and TGController	88
3.5	Database Schema Migrations	91
3.6	Testing with TurboGears	93
3.7	TurboGears2 Configuration	98
3.8	Authentication in TurboGears 2 applications	105
3.9	Hooks and Wrappers	109
4	TurboGears2 CookBook	113
4.1	Upgrading TurboGears	113
4.2	Basic Recipes	122
4.3	Advanced Recipes	163
4.4	Contributing to TurboGears	187

5 TurboGears Reference	193
5.1 Configuration Options	193
5.2 Classes and Functions	195
6 The TurboGears documentation	223
7 Getting Started	225
7.1 Installing TurboGears	225
7.2 Single File Application	225
7.3 Full Stack Projects	226
Python Module Index	227

This section covers a bunch of tutorials and documentation about getting started with TurboGears2, the *Basic Documentation* section will cover documentation for people that are approaching TurboGears for the first time, while the *Advanced Documentation* chapter will provide documentation for people that want to go down into the framework or are having more complex needs.

1.1 Hello TurboGears

The fastest way to start using TurboGears is through the **minimal mode**, when using TurboGears with minimal mode a default setup that minimizes dependencies and complexity is provided.

Note: While minimal mode is well suited for small simple web applications or web services, for more complex projects moving to a package based configuration is suggested. To start with a package based application the *20 Minutes Wiki Tutorial* tutorial is provided.

1.1.1 Play with TurboGears

If you want to experiment with this tutorial without installing TurboGears on your local machine, feel free to edit the [Basic TurboGears Example](#) on Runnable and skip the *Setup* section.

This will provide a working TurboGears application in your browser you can freely edit and run.

1.1.2 Setup

First we are going to create a virtual environment where to install the framework, if you want to proceed without using a virtual environment simply skip to *Install TurboGears*. Keep in mind that using a virtual environment is the suggested way to install TurboGears without messing with your system packages and python modules. To do so we need to install the `virtualenv` package:

```
$ pip install virtualenv
```

Now the `virtualenv` command should be available and we can create and activate a virtual environment for our TurboGears2 project:

```
$ virtualenv tgenv
$ . tgenv/bin/activate
```

If our environment got successfully created and activated we should end up with a prompt that looks like:

```
(tgenv)$
```

Now we are ready to install TurboGears itself:

```
(tgenv)$ pip install TurboGears2
```

1.1.3 Hello World

A TurboGears application consists of an `AppConfig` application configuration and an application `RootController`. The first is used to setup and create the application itself, while the latter is used to dispatch requests and take actions.

For our first application we are going to define a controller with an `index` method that just tells *Hello World*:

```
from tg import expose, TGController, AppConfig

class RootController(TGController):
    @expose()
    def index(self):
        return 'Hello World'
```

now to make TurboGears serve our controller we must create the actual application from an `AppConfig`:

```
config = AppConfig(minimal=True, root_controller=RootController())

application = config.make_wsgi_app()
```

then we must actually serve the application:

```
from wsgiref.simple_server import make_server

print "Serving on port 8080..."
httpd = make_server('', 8080, application)
httpd.serve_forever()
```

Running the Python module just created will start a server on port 8080 with the our hello world application, opening your browser and pointing it to `http://localhost:8080` should present you with an `Hello World` text.

1.1.4 Greetings

Now that we have a working application it's time to say hello to our user instead of greeting the world, to do so we can extend our controller with an `hello` method which gets as a parameter the person to greet:

```
class RootController(TGController):
    @expose()
    def index(self):
        return 'Hello World'

    @expose()
    def hello(self, person):
        return 'Hello %s' % person
```

Restarting the application and pointing the browser to `http://localhost:8080/hello?person=MyName` should greet you with an **Hello MyName** text.

Note: How and why requests are routed to the `index` and `hello` methods is explained in *Object Dispatch* documentation

Passing parameters to your controllers is as simple as adding them to the url with the same name of the parameters in your method, TurboGears will automatically map them to function arguments when calling an exposed method.

1.1.5 Serving Templates

Being able to serve text isn't usually enough for a web application, for more advanced output using a template is usually preferred. Before being able to serve a template we need to install a template engine and enable it.

The template engine we are going to use for this example is Jinja2 which is a fast and flexible template engine with python3 support. To install jinja simply run:

```
(tgenv)$ pip install jinja2
```

Now that the template engine is available we need to enable it in TurboGears, doing so is as simple as adding it to the list of the available engines inside our AppConfig:

```
config = AppConfig(minimal=True, root_controller=RootController())
config.renderers = ['jinja']
```

```
application = config.make_wsgi_app()
```

Now our application is able to expose templates based on the Jinja template engine, to test them we are going to create an `hello.jinja` file inside the same directory where our application is available:

```
<!doctype html>
<title>Hello</title>
{% if person %}
  <h1>Hello {{ person }}</h1>
{% else %}
  <h1>Hello World!</h1>
{% endif %}
```

then the `hello` method will be changed to display the newly created template instead of using a string directly:

```
class RootController(TGController):
    @expose()
    def index(self):
        return 'Hello World'

    @expose('hello.jinja')
    def hello(self, person=None):
        return dict(person=person)
```

Restarting the application and pointing the browser to `http://localhost:8080/hello` or `http://localhost:8080/hello?person=MyName` will display an `hello` page greeting the person whose name is passed as parameter or the world itself if the parameter is missing.

1.1.6 Serving Statics

Even for small web applications being able to apply style through CSS or serving javascript scripts is often required, to do so we must tell TurboGears to serve our static files and from where to serve them:

```
config = AppConfig(minimal=True, root_controller=RootController())
config.renderers = ['jinja']
config.serve_static = True
config.paths['static_files'] = 'public'

application = config.make_wsgi_app()
```

After restating the application, any file placed inside the `public` directory will be served directly by TurboGears. Supposing you have a `style.css` file you can access it as `http://localhost:8080/style.css`.

1.1.7 Going Forward

While it is possible to manually enable more advanced features like the `SQLAlchemy` and `Ming` storage backends, the application helpers, `app_globals`, `i18n` and all the TurboGears features through the `AppConfig` object, if you need them you probably want TurboGears to create a full featured application through the `gearbox quickstart` command.

The *20 Minutes Wiki Tutorial* provides an introduction to more complex applications enabled all the TurboGears features, follow it if you want to unleash all the features that TurboGears provides!

1.2 Full Featured TurboGears: A Wiki in 20 Minutes

How does TurboGears2 help you get development done quickly? We'll show you by developing a simple wiki application that should take you no more than 20 minutes to complete. We're going to do this without explaining the steps in detail (that is what this book is for, after all). As a result, you'll see how easily you can make your own web applications once you are up to speed on what TurboGears2 offers.

If you're not familiar with the concept of a wiki you might want to check out [the Wikipedia entry](#). Basically, a wiki is an easily-editable collaborative web content system that makes it trivial to link to pages and create new pages. Like other wiki systems, we are going to use CamelCase words to designate links to pages.

If you have trouble with this tutorial ask for help on the [TurboGears discussion list](#), or on the IRC channel `#turbogears`. We're a friendly bunch and, depending what time of day you post, you'll get your answer in a few minutes to a few hours. If you search the mailing list or the web in general you'll probably get your answer even faster. **Please don't post your problem reports as comments on this or any of the following pages of the tutorial.** Comments are for suggestions for improvement of the docs, not for seeking support.

If you want to see the final version you can download a copy of the wiki code.

1.2.1 Setup

This tutorial takes for granted that you have a working Python environment with Python2.6 or Python2.7, with `pip` installed and you have a working browser to look at the web application you are developing.

This tutorial doesn't cover Python at all. Check the [Python Documentation](#) page for more coverage of Python.

Setting up our Environment

If it is your first TurboGears2 project you need to create an environment and install the TurboGears2 web framework to make the development commands available.

Creating the Environment

First we are going to create a Virtual Environment where to install the framework, this helps keeping our system clean by not installing the packages system-wide. To do so we need to install the `virtualenv` package:

```
$ pip install virtualenv
```

Now the `virtualenv` command should be available and we can create and activate a virtual environment for our TurboGears2 project:

```
$ virtualenv tgenv
$ . tgenv/bin/activate
```

If our environment got successfully created and activated we should end up with a prompt that looks like:

```
(tgenv)$
```

Installing TurboGears2

TurboGears2 can be quickly installed by installing the TurboGears2 development tools, those will install TurboGears2 itself and a bunch of commands useful when developing TurboGears applications:

```
(tgenv)$ pip install tg.devtools
```

Note: The *-i http://tg.gy/VERSION* option is used to make sure that we install TurboGears2 latest version and its dependencies at the right version, replacing it, for example, with 220 or 215 will install the 2.2 and 2.1.5 version respectively. TurboGears2 package doesn't usually enforce dependencies version to make possible for developers to upgrade dependencies if they need a bugfix or new features. It is suggested to always use the *-i* option to avoid installing incompatible packages.

1.2.2 Quickstart

TurboGears2 provides a suite of tools for working with projects by adding several commands to the Python command line tool *gearbox*. A few will be touched upon in this tutorial. (Check the *GearBox* section for a full listing.) The first tool you'll need is *quickstart*, which initializes a TurboGears project. Go to a command line window and run the following command:

```
(tgenv)$ gearbox quickstart wiki20
```

This will create a project called *wiki20* with the default template engine and with authentication. TurboGears2 projects usually share a common structure, which should look like:

```
wiki20
-- __init__.py
-- config <-- Where project setup and configuration relies
-- controllers <-- All the project controllers, the logic of our web application
-- i18n <-- Translation files for the languages supported
-- lib <-- Utility python functions and classes
-- model <-- Database models
-- public <-- Static files like CSS, javascript and images
-- templates <-- Templates exposed by our controllers.
-- tests <-- Tests
-- websetup <-- Functions to execute at application setup. Like creating tables, a standard user
```

Note: We recommend you use the names given here: this documentation looks for files in directories based on these names.

You need to update the dependencies in the file *Wiki-20/setup.py*. Look for a list named *install_requires* and append the *docutils* entry at the end. TurboGears2 does not require *docutils*, but the wiki we are building does.

Your *install_requires* should end up looking like:

```
install_requires=[
    "TurboGears2 >= 2.3.0",
    "Genshi",
    "zope.sqlalchemy >= 0.4",
    "sqlalchemy",
    "sqlalchemy-migrate",
```

```
"repoze.who",
"repoze.who-friendlyform >= 1.0.4",
"tgext.admin >= 0.5.1",
"repoze.who.plugins.sa",
"tw2.forms",
"docutils"
]
```

Now to be able to run the project you will need to install it and its dependencies. This can be quickly achieved by running from inside the `wiki20` directory:

```
$ pip install -e .
```

Note: If you skip the `pip install -e .` command you might end up with an error that looks like: *pkg_resources.DistributionNotFound: tw2.forms: Not Found for: wiki20 (did you run python setup.py develop?)* This is because some of the dependencies your project depend on the options you choose while quickstarting it.

You should now be able to start the newly create project with the `gearbox serve` command:

```
(tgenv)$ gearbox serve --reload
Starting subprocess with file monitor
Starting server in PID 32797.
serving on http://127.0.0.1:8080
```

Note: The `--reload` option makes the server restart whenever a file is changed, this greatly speeds up the development process by avoiding to manually restart the server whenever we need to try our changes.

Pointing your browser to <http://127.0.0.1:8080/> should open up the TurboGears2 welcome page. By default newly quickstarted projects provide a bunch of pages to guide the user through some of the foundations of TurboGears2 web applications.

1.2.3 Controller And View

TurboGears follows the [Model-View-Controller paradigm](#) (a.k.a. “MVC”), as do most modern web frameworks like Rails, Django, Struts, etc.

Taking a look at the <http://127.0.0.1:8080/about> page is greatly suggested to get an overview of your newly quickstarted project and how TurboGears2 works.

If you take a look at the code that `quickstart` created, you’ll see everything necessary to get up and running. Here, we’ll look at the two files directly involved in displaying this welcome page.

Controller Code

`Wiki-20/wiki20/controllers/root.py` (see below) is the code that causes the welcome page to be produced. After the imports the first line of code creates our main controller class by inheriting from TurboGears’ `BaseController`:

```
class RootController(BaseController):
```

The TurboGears 2 controller is a simple object publishing system; you write controller methods and `@expose()` them to the web. In our case, there’s a single controller method called `index`. As you might guess, this name is not accidental; this becomes the default page you’ll get if you go to this URL without specifying a particular destination, just like you’ll end up at `index.html` on an ordinary web server if you don’t give a specific file name. You’ll

also go to this page if you explicitly name it, with `http://localhost:8080/index`. We'll see other controller methods later in the tutorial so this naming system will become clear.

The `@expose()` decorator tells TurboGears which template to use to render the page. Our `@expose()` specifies:

```
@expose('wiki20.templates.index')
```

This gives TurboGears the template to use, including the path information (the `.html` extension is implied). We'll look at this file shortly.

Each controller method returns a dictionary, as you can see at the end of the `index` method. TG takes the key:value pairs in this dictionary and turns them into local variables that can be used in the template.

```
from tg import expose, flash, require, url, request, redirect
#Skipping some imports here...

class RootController(BaseController):
    secc = SecureController()
    admin = AdminController(model, DBSession, config_type=TGAdminConfig)

    error = ErrorController()

    def _before(self, *args, **kw):
        tmpl_context.project_name = "Wiki 20"

    @expose('wiki20.templates.index')
    def index(self):
        """Handle the front-page."""
        return dict(page='index')

    #more controller methods from here on...
```

Displaying The Page

`Wiki-20/wiki20/templates/index.html` is the template specified by the `@expose()` decorator, so it formats what you view on the welcome screen. Look at the file; you'll see that it's standard XHTML with some simple namespaced attributes. This makes it very designer-friendly, and well-behaved design tools will respect all the Genshi attributes and tags. You can even open it directly in your browser.

Genshi directives are elements and/or attributes in the template that are usually prefixed with `py:`. They can affect how the template is rendered in a number of ways: Genshi provides directives for conditionals and looping, among others. We'll see some simple Genshi directives in the sections on *Editing pages* and *Adding views*.

The following is the content of a newly quickstarted TurboGears2 project at 2.2 release time:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:py="http://genshi.edgewall.org/"
      xmlns:xi="http://www.w3.org/2001/XInclude">

    <xi:include href="master.html" />

<head>
    <title>Welcome to TurboGears 2.2, standing on the shoulders of giants, since 2007</title>
</head>

<body>
    <div class="row">
        <div class="span8 hidden-phone hidden-tablet">
            <div class="hero-unit">
```

```
<h1>Welcome to TurboGears 2.2</h1>
<p>If you see this page it means your installation was successful!</p>
<p>TurboGears 2 is rapid web application development toolkit designed to make your life easier.
<p>
  <a class="btn btn-primary btn-large" href="http://www.turbogears.org" target="_blank">
    ${h.icon('book', True)} Learn more
  </a>
</p>
</div>
</div>
<div class="span4">
  <a class="btn btn-small" href="http://www.turbogears.org/2.2/docs/">${h.icon('book')} TG2 Documentation
  <span class="label label-success">new</span>
  Read the Getting Started section<br/>
  <br/>
  <a class="btn btn-small" href="http://www.turbogears.org/book/">${h.icon('book')} TG2 Book</a>
  Work in progress TurboGears2 book<br/>
  <br/>
  <a class="btn btn-small" href="http://groups.google.com/group/turbogears">${h.icon('comment')}
  for general TG use/topics
</div>
</div>

<div class="row">
  <div class="span4">
    <h3>Code your data model</h3>
    <p> Design your data model, Create the database, and Add some bootstrap data.</p>
  </div>
  <div class="span4">
    <h3>Design your URL architecture</h3>
    <p> Decide your URLs, Program your controller methods, Design your
    templates, and place some static files (CSS and/or Javascript). </p>
  </div>
  <div class="span4">
    <h3>Distribute your app</h3>
    <p> Test your source, Generate project documents, Build a distribution.</p>
  </div>
</div>

<div class="notice"> Thank you for choosing TurboGears.</div>
</body>
</html>
```

1.2.4 Wiki Model

quickstart produced a directory for our model in `Wiki-20/wiki20/model/`. This directory contains an `__init__.py` file, which makes that directory name into a python module (so you can use `import model`).

Since a wiki is basically a linked collection of pages, we'll define a Page class as the name of our model.

Create a new file called `Wiki-20/wiki20/model/page.py`:

```
from sqlalchemy import *
from sqlalchemy.orm import mapper, relation
from sqlalchemy import Table, ForeignKey, Column
from sqlalchemy.types import Integer, Text

from wiki20.model import DeclarativeBase, metadata, DBSession
```

```
class Page(DeclarativeBase):
    __tablename__ = 'page'

    id = Column(Integer, primary_key=True)
    pagename = Column(Text, unique=True)
    data = Column(Text)
```

Now to let TurboGears know that our model exists we must make it available inside the Wiki-20/wiki20/model/___init___ .py file just by importing it at the end:

```
# Import your model modules here.
from wiki20.model.auth import User, Group, Permission
from wiki20.model.page import Page
```

Warning: It's very important that this line is at the end because Page requires the rest of the model to be initialized before it can be imported:

Initializing The Tables

Now that our model is recognized by TurboGears we must create the table that it is going to use to store its data. By default TurboGears will automatically create tables for each model it is aware of, this is performed during the application setup phase.

The setup phase is managed by the Wiki-20/wiki20/websetup python module, we are just going to add to "websetup/boostrap.py" the lines required to create a FrontPage page for our wiki, so it doesn't start empty.

We need to update the file to create our *FrontPage* data just before the `DBSession.flush()` command by adding:

```
page = model.Page(pagename="FrontPage", data="initial data")
model.DBSession.add(page)
```

You should end up having a `try:except:` block that should look like:

```
def bootstrap(command, conf, vars):
    #Some comments and setup here...

    try:
        #Users and groups get created here...
        model.DBSession.add(u1)

        page = model.Page(pagename="FrontPage", data="initial data")
        model.DBSession.add(page)

        model.DBSession.flush()
        transaction.commit()
    except IntegrityError:
        #Some Error handling here...
```

The `transaction.commit()` call involves the transaction manager used by TurboGears2 which helps us to support cross database transactions, as well as transactions in non relational databases.

Now to actually create our table and our *FrontPage* we simply need to run the `gearbox setup-app` command where your application configuration file is available (usually the root of the project):

```
(tgenv)$ gearbox setup-app
Running setup_app() from wiki20.websetup
Creating tables
```

A file named `Wiki-20/devdata.db` should be created which contains your `sqlite` database. For other database systems refer to the `sqlalchemy.url` line inside your configuration file.

1.2.5 Adding Controllers

Controllers are the code that figures out which page to display, what data to grab from the model, how to process it, and finally hands off that processed data to a template.

`quickstart` has already created some basic controller code for us at `Wiki-20/wiki20/controllers/root.py`.

First, we must import the `Page` class from our model. At the end of the `import` block, add this line:

```
from wiki20.model.page import Page
```

Now we will change the template used to present the data, by changing the `@expose('wiki20.templates.index')` line to:

```
@expose('wiki20.templates.page')
```

This requires us to create a new template named `page.html` in the `wiki20/templates` directory; we'll do this in the next section.

Now we must specify which page we want to see. To do this, add a parameter to the `index()` method. Change the line after the `@expose` decorator to:

```
def index(self, pagename="FrontPage"):
```

This tells the `index()` method to accept a parameter called `pagename`, with a default value of `"FrontPage"`.

Now let's get that page from our data model. Put this line in the body of `index`:

```
page = DBSession.query(Page).filter_by(pagename=pagename).one()
```

This line asks the SQLAlchemy database session object to run a query for records with a `pagename` column equal to the value of the `pagename` parameter passed to our controller method. The `.one()` method assures that there is only one returned result; normally a `.query` call returns a list of matching objects. We only want one page, so we use `.one()`.

Finally, we need to return a dictionary containing the `page` we just looked up. When we say:

```
return dict(wikipage=page)
```

The returned `dict` will create a template variable called `wikipage` that will evaluate to the `page` object that we looked it up.

Your `index` controller method should end up looking like:

```
from tg import expose, flash, require, url, request, redirect
```

```
#More imports here...
```

```
from wiki20.model.page import Page
```

```
class RootController(BaseController):
```

```
    secc = SecureController()
```

```
    admin = AdminController(model, DBSession, config_type=TGAdminConfig)
```

```
    error = ErrorController()
```

```
    def _before(self, *args, **kw):
```

```

    tpl_context.project_name = "Wiki 20"

    @expose('wiki20.templates.page')
    def index(self, pagename="FrontPage"):
        page = DBSession.query(Page).filter_by(pagename=pagename).one()
        return dict(wikipage=page)

    #more controller methods from here on...

```

Now our `index()` method fetches a record from the database (creating an instance of our mapped `Page` class along the way), and returns it to the template within a dictionary.

1.2.6 Adding Views (Templates)

quickstart also created some templates for us in the `Wiki-20/wiki20/templates` directory: `master.html` and `index.html`. Back in our simple controller, we used `@expose()` to hand off a dictionary of data to a template called `'wiki20.templates.index'`, which corresponds to `Wiki-20/wiki20/templates/index.html`.

Take a look at the following line in `index.html`:

```
<xi:include href="master.html" />
```

This tells the `index` template to *include* the `master` template. Using includes lets you easily maintain a cohesive look and feel throughout your site by having each page include a common master template.

Copy the contents of `index.html` into a new file called `page.html`. Now modify it for our purposes:

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:py="http://genshi.edgewall.org/"
    xmlns:xi="http://www.w3.org/2001/XInclude">

    <xi:include href="master.html" />

<head>
    <meta content="text/html; charset=UTF-8" http-equiv="content-type" py:replace="''"/>
    <title>${wikipage.pagename} - The TurboGears 2 Wiki</title>
</head>

<body>
    <div class="main_content">
        <div style="float:right; width: 10em;"> Viewing
            <span py:replace="wikipage.pagename">Page Name Goes Here</span>
            <br/>
            You can return to the <a href="/">FrontPage</a>.
        </div>

        <div py:replace="wikipage.data">Page text goes here.</div>

        <div>
            <a href="/edit/${wikipage.pagename}">Edit this page</a>
        </div>
    </div>
</body>
</html>

```

This is a basic XHTML page with three substitutions:

1. In the `<title>` tag, we substitute the name of the page, using the `pagename` value of `page`. (Remember, `wikipedia` is an instance of our mapped `Page` class, which was passed in a dictionary by our controller.):

```
<title>${wikipedia.pagename} - The TurboGears 2 Wiki</title>
```

2. In the second `<div>` element, we substitute the page name again with Genshi's `py:replace`:

```
<span py:replace="wikipedia.pagename">Page Name Goes Here</span>
```

3. In the third `<div>`, we put in the contents of our “`wikipedia`”:

```
<div py:replace="wikipedia.data">Page text goes here.</div>
```

When you refresh the output web page you should see “initial data” displayed on the page.

Note: `py.replace` replaces the *entire tag* (including start and end tags) with the value of the variable provided.

1.2.7 Editing pages

One of the fundamental features of a wiki is the ability to edit the page just by clicking “Edit This Page,” so we’ll create a template for editing. First, make a copy of `page.html`:

```
cd wiki20/templates
cp page.html edit.html
```

We need to replace the content with an editing form and ensure people know this is an editing page. Here are the changes for `edit.html`.

1. Change the title in the header to reflect that we are editing the page:

```
<head>
  <meta content="text/html; charset=UTF-8" http-equiv="content-type" py:replace="''"/>
  <title>Editing: ${wikipedia.pagename}</title>
</head>
```

2. Change the div that displays the page:

```
<div py:replace="wikipedia.data">Page text goes here.</div>
```

with a div that contains a standard HTML form:

```
<div>
  <form action="/save" method="post">
    <input type="hidden" name="pagename" value="${wikipedia.pagename}"/>
    <textarea name="data" py:content="wikipedia.data" rows="10" cols="60"/>
    <input type="submit" name="submit" value="Save"/>
  </form>
</div>
```

Now that we have our view, we need to update our controller in order to display the form and handle the form submission. For displaying the form, we’ll add an `edit` method to our controller in `Wiki-20/wiki20/controllers/root.py`:

```
from tg import expose, flash, require, url, request, redirect
```

```
#More imports here...
```

```
from wiki20.model.page import Page
```

```

class RootController(BaseController):
    secc = SecureController()
    admin = AdminController(model, DBSession, config_type=TGAdminConfig)

    error = ErrorController()

    def _before(self, *args, **kw):
        tmpl_context.project_name = "Wiki 20"

    @expose('/wiki20.templates.page')
    def index(self, pagename="FrontPage"):
        page = DBSession.query(Page).filter_by(pagename=pagename).one()
        return dict(wikipage=page)

    @expose(template="wiki20.templates.edit")
    def edit(self, pagename):
        page = DBSession.query(Page).filter_by(pagename=pagename).one()
        return dict(wikipage=page)

    #more controller methods from here on...

```

For now, the new method is identical to the `index` method; the only difference is that the resulting dictionary is handed to the `edit` template. To see it work, go to <http://localhost:8080/edit/FrontPage>. However, this only works because `FrontPage` already exists in our database; if you try to edit a new page with a different name it will fail, which we'll fix in a later section.

Don't click that save button yet! We still need to write that method.

1.2.8 Saving Our Edits

When we displayed our wiki's edit form in the last section, the form's `action` was `/save`. So, we need to make a method called `save` in the `Root` class of our controller.

However, we're also going to make another important change. Our `index` method is *only* called when you either go to `/` or `/index`. If you change the `index` method to the special method `_default`, then `_default` will be automatically called whenever nothing else matches. `_default` will take the rest of the URL and turn it into positional parameters. This will cause the wiki to become the default when possible.

Here's our new version of `root.py` which includes both `_default` and `save`:

```

from tg import expose, flash, require, url, request, redirect

#More imports here...

from wiki20.model.page import Page

class RootController(BaseController):
    secc = SecureController()
    admin = AdminController(model, DBSession, config_type=TGAdminConfig)

    error = ErrorController()

    def _before(self, *args, **kw):
        tmpl_context.project_name = "Wiki 20"

    @expose('/wiki20.templates.page')
    def _default(self, pagename="FrontPage"):

```

```
    """Handle the front-page."""
    page = DBSession.query(Page).filter_by(pagename=pagename).one()
    return dict(wikipage=page)

@expose(template="wiki20.templates.edit")
def edit(self, pagename):
    page = DBSession.query(Page).filter_by(pagename=pagename).one()
    return dict(wikipage=page)

@expose()
def save(self, pagename, data, submit):
    page = DBSession.query(Page).filter_by(pagename=pagename).one()
    page.data = data
    redirect("/") + pagename)

#more controller methods from here on...
```

Unlike the previous methods we've made, `save` just uses a plain `@expose()` without any template specified. That's because we're only redirecting the user back to the viewing page.

Although the `page.data = data` statement tells SQLAlchemy that you intend to store the page data in the database, you would usually need to flush the SQLAlchemy Unit of Work and commit the currently running transaction, those are operations that TurboGears2 transaction management will automatically do for us.

You don't have to do anything to use this transaction management system, it should just work. So, you can now make changes and save the page we were editing, just like a real wiki.

1.2.9 What About WikiWords?

Our wiki doesn't yet have a way to link pages. A typical wiki will automatically create links for *WikiWords* when it finds them (WikiWords have also been described as *WordsSmashedTogether*). This sounds like a job for a regular expression.

Here's the new version of our `RootController._default` method, which will be explained afterwards:

```
from tg import expose, flash, require, url, request, redirect

#More imports here...

from wiki20.model.page import Page
import re
from docutils.core import publish_parts

wikiwords = re.compile(r"\b([A-Z]\w+[A-Z]+\w+)")

class RootController(BaseController):
    secc = SecureController()
    admin = AdminController(model, DBSession, config_type=TGAdminConfig)

    error = ErrorController()

    def _before(self, *args, **kw):
        tmpl_context.project_name = "Wiki 20"

    @expose('wiki20.templates.page')
    def _default(self, pagename="FrontPage"):
        page = DBSession.query(Page).filter_by(pagename=pagename).one()
        content = publish_parts(page.data, writer_name="html")["html_body"]
```

```

root = url('/')
content = wikiwords.sub(r'<a href="%s\1">\1</a>' % root, content)
return dict(content=content, wikipage=page)

@expose(template="wiki20.templates.edit")
def edit(self, pagename):
    page = DBSession.query(Page).filter_by(pagename=pagename).one()
    return dict(wikipage=page)

@expose()
def save(self, pagename, data, submit):
    page = DBSession.query(Page).filter_by(pagename=pagename).one()
    page.data = data
    redirect("/") + pagename)

#more controller methods from here on...

```

We need some additional imports, including `re` for regular expressions and a method called `publish_parts` from `docutils`.

A WikiWord is a word that starts with an uppercase letter, has a collection of lowercase letters and numbers followed by another uppercase letter and more letters and numbers. The `wikiwords` regular expression describes a WikiWord.

In `_default`, the new lines begin with the use of `publish_parts`, which is a utility that takes string input and returns a dictionary of document parts after performing conversions; in our case, the conversion is from Restructured Text to HTML. The input (`page.data`) is in Restructured Text format, and the output format (specified by `writer_name="html"`) is in HTML. Selecting the `fragment` part produces the document without the document title, subtitle, docinfo, header, and footer.

You can configure TurboGears so that it doesn't live at the root of a site, so you can combine multiple TurboGears apps on a single server. Using `tg.url()` creates relative links, so that your links will continue to work regardless of how many apps you're running.

The next line rewrites the `content` by finding any WikiWords and substituting hyperlinks for those WikiWords. That way when you click on a WikiWord, it will take you to that page. The `r' string'` means 'raw string', one that turns off escaping, which is mostly used in regular expression strings to prevent you from having to double escape slashes. The substitution may look a bit weird, but is more understandable if you recognize that the `%s` gets substituted with `root`, then the substitution is done which replaces the `\1` with the string matching the regex.

Note that `_default()` is now returning a `dict` containing an additional key-value pair: `content=content`. This will not break `wiki20.templates.page` because that page is only looking for `page` in the dictionary, however if we want to do something interesting with the new key-value pair we'll need to edit `wiki20.templates.page`:

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:py="http://genshi.edgewall.org/"
      xmlns:xi="http://www.w3.org/2001/XInclude">

    <xi:include href="master.html" />

<head>
    <meta content="text/html; charset=UTF-8" http-equiv="content-type" py:replace="" />
    <title>${wikipage.pagename} - The TurboGears 2 Wiki</title>
</head>

<body>
    <div class="main_content">

```

```
<div style="float:right; width: 10em;"> Viewing
  <span py:replace="wikipage.pagename">Page Name Goes Here</span>
  <br/>
  You can return to the <a href="/">FrontPage</a>.
</div>

<div py:replace="Markup(content)">Formatted content goes here.</div>

<div>
  <a href="/edit/${wikipage.pagename}">Edit this page</a>
</div>
</div>
</body>
</html>
```

Since `content` comes through as XML, we can strip it off using the `Markup()` function to produce plain text (try removing the function call to see what happens).

To test the new version of the system, edit the data in your front page to include a WikiWord. When the page is displayed, you'll see that it's now a link. You probably won't be surprised to find that clicking that link produces an error.

1.2.10 Hey, Where's The Page?

What if a Wiki page doesn't exist? We'll take a simple approach: if the page doesn't exist, you get an edit page to use to create it.

In the `_default` method, we'll check to see if the page exists.

If it doesn't, we'll redirect to a new `notfound` method. We'll add this method after the `_default` method and before the `edit` method.

Here are the new `notfound` and the updated `_default` methods for our `RootController` class:

```
@expose('wiki20.templates.page')
def _default(self, pagename="FrontPage"):
    from sqlalchemy.exc import InvalidRequestError

    try:
        page = DBSession.query(Page).filter_by(pagename=pagename).one()
    except InvalidRequestError:
        raise redirect("notfound", pagename=pagename)

    content = publish_parts(page.data, writer_name="html")["html_body"]
    root = url('/')
    content = wikiwords.sub(r'<a href="%s\1">\1</a>' % root, content)
    return dict(content=content, wikipage=page)

@expose("wiki20.templates.edit")
def notfound(self, pagename):
    page = Page(pagename=pagename, data="")
    DBSession.add(page)
    return dict(wikipage=page)
```

In the `_default` code we now first try to get the page and then deal with the exception by redirecting to a method that will make a new page.

As for the `notfound` method, the first two lines of the method add a row to the page table. From there, the path is exactly the same it would be for our `edit` method.

With these changes in place, we have a fully functional wiki. Give it a try! You should be able to create new pages now.

1.2.11 Adding A Page List

Most wikis have a feature that lets you view an index of the pages. To add one, we'll start with a new template, *pagelist.html*. We'll copy *page.html* so that we don't have to write the boilerplate.

```
cd wiki20/templates
cp page.html pagelist.html
```

After editing, our *pagelist.html* looks like:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:py="http://genshi.edgewall.org/"
      xmlns:xi="http://www.w3.org/2001/XInclude">

    <xi:include href="master.html" />

<head>
    <meta content="text/html; charset=UTF-8" http-equiv="content-type" py:replace="''"/>
    <title>Page Listing - The TurboGears 2 Wiki</title>
</head>

<body>
    <div class="main_content">
        <h1>All Pages</h1>
        <ul>
            <li py:for="pagename in pages">
                <a href="{tg.url('/' + pagename)}"
                    py:content="pagename">
                    Page Name Here.
                </a>
            </li>
        </ul>
        Return to the <a href="/">FrontPage</a>.
    </div>
</body>
</html>
```

The highlighted section represents the Genshi code of interest. You can guess that the `py:for` is a python `for` loop, modified to fit into Genshi's XML. It iterates through each of the `pages` (which we'll send in via the controller, using a modification you'll see next). For each one, `Page Name Here` is replaced by `pagename`, as is the URL. You can learn more about the [Genshi templating engine](#) at their site.

We must also modify the `RootController` class to implement `pagelist` and to create and pass `pages` to our template:

```
@expose("wiki20.templates.pagelist")
def pagelist(self):
    pages = [page.pagename for page in DBSession.query(Page).order_by(Page.pagename)]
    return dict(pages=pages)
```

Here, we select all of the `Page` objects from the database, and order them by `pagename`.

We can also modify *page.html* so that the link to the page list is available on every page:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:py="http://genshi.edgewall.org/"
    xmlns:xi="http://www.w3.org/2001/XInclude">

    <xi:include href="master.html" />

<head>
    <meta content="text/html; charset=UTF-8" http-equiv="content-type" py:replace="''"/>
    <title>${wikipage.pagename} - The TurboGears 2 Wiki</title>
</head>

<body>
    <div class="main_content">
        <div style="float:right; width: 10em;"> Viewing
            <span py:replace="wikipage.pagename">Page Name Goes Here</span>
            <br/>
            You can return to the <a href="/">FrontPage</a>.
        </div>

        <div py:replace="Markup(content)">Formatted content goes here.</div>

        <div>
            <a href="/edit/${wikipage.pagename}">Edit this page</a>
            <a href="/pagelist">View the page list</a>
        </div>
    </div>
</body>
</html>
```

You can see your pagelist by clicking the link on a page or by going directly to <http://localhost:8080/pagelist>.

1.2.12 Further Exploration

Now that you have a working Wiki, there are a number of further places to explore:

1. You can add JSON support via jQuery
2. You can learn more about the [Genshi templating engine](#).
3. You can learn more about the [SQLAlchemy ORM](#).

Todo

Add link to help show how to add jQuery support

If you had any problems with this tutorial, or have ideas on how to make it better, please let us know on the [mailing list](#)! Suggestions are almost always incorporated.

1.3 Rapid Prototyping: A Wiki using the TurboGears Admin

TurboGears2 Wikier application is inspired by the *TurboGears2 20 Minutes Wiki* Tutorial.

While the 20 Minutes Wiki is a great way to learn writing custom TurboGears2 applications the Wikier tutorial tries to provide more focus on rapid prototyping with the ambitious target of making it possible to create a full featured Wiki

in minimum possible time.

It will showcase some basic concepts of the TurboGears2 Web Framework and how to best use rapid prototyping tools available in the standard extension modules.

Contents:

1.3.1 Creating Project Structure

Hint: This tutorial has been written for TurboGears 2.3 on Python2.7. While it might work with previous or later versions, it has been tested only for version 2.3.

Setting up our Environment

If this is your first TurboGears2 project you need to create an environment and install the TurboGears2 web framework to make the development commands available.

Completed version of this tutorial is available on <http://runnable.com/U8P0CQTKHwNzQoYs/turbogears-wikier-tutorial-for-python>.

If you want to play around with this tutorial without installing TurboGears on your computer you can freely edit the *Runnable* version.

Creating the Environment

First we are going to create a Virtual Environment where we will install the framework. This helps keeping our system clean by not installing the packages system-wide. To do so we need to install the `virtualenv` package:

```
$ pip install virtualenv
```

Now the `virtualenv` command should be available and we can create and activate a virtual environment for our TurboGears2 project:

```
$ virtualenv tgenv
$ . tgenv/bin/activate
```

If our environment got successfully created and activated we should end up with a prompt that looks like:

```
(tgenv) $
```

Installing TurboGears2

TurboGears2 can be quickly installed by installing the TurboGears2 development tools. This will install TurboGears2 itself and a bunch of commands useful when developing TurboGears applications:

```
(tgenv) $ pip install tg.devtools
```

Note: The `-i http://tg.gy/VERSION` option is used to make sure that we install TurboGears2 version and its dependencies at the right version. TurboGears2 doesn't usually enforce version dependencies to make it possible for developers to upgrade them if they need a bugfix or new features. It is suggested to always use the `-i` option to avoid installing incompatible packages.

Creating the Project

If the install correctly completed the `gearbox quickstart` command should be available in your virtual environment:

```
(tgenvenv)$ gearbox quickstart wikir
```

This will create a project called `wikir` with the default template engine and with authentication. TurboGears2 projects usually share a common structure, which should look like:

```
wikir
-- __init__.py
-- config      <-- Where project setup and configuration is located
-- controllers <-- All the project controllers, the logic of our web application
-- i18n        <-- Translation files for the languages supported
-- lib         <-- Utility python functions and classes
-- model       <-- Database models
-- public      <-- Static files like CSS, javascript and images
-- templates   <-- Templates exposed by our controllers
-- tests       <-- Tests
-- websetup    <-- Functions to execute at application setup like creating tables, a standard user a
```

Installing Project and its Dependencies

Before we can start our project and open it into a browser we must install any dependency that is not strictly related to TurboGears itself. This can easily be achieved running the `develop` command which will install into our environment the project itself and all its dependencies:

```
(tgenvenv)$ cd wikir
(tgenvenv)$ pip install -e .
```

Project dependencies are specified inside the `setup.py` file in the `install_requires` list. Default project dependencies should look like:

```
install_requires=[
    "TurboGears2 >= 2.3.0",
    "Genshi",
    "zope.sqlalchemy >= 0.4",
    "sqlalchemy",
    "sqlalchemy-migrate",
    "repoze.who",
    "tgext.admin >= 0.5.1",
    "repoze.who.plugins.sa",
    "tw2.forms",
]
```

Genshi dependency is the template engine our application is going to use, the *zope.sqlalchemy*, *sqlalchemy* and *sqlalchemy-migrate* dependencies are there to provide support for SQLALchemy based database layer. *repoze.who* and *repoze.who.plugins.sa* are used by the authentication and authorization layer. *tgext.admin* and *tw2.forms* are used to generate administrative interfaces and forms.

Serving our Project

Note: If you skipped the `pip install -e .` command you might end up with an error that looks like: *pkg_resources.DistributionNotFound: tw2.forms: Not Found for: wikir (did you run python setup.py develop?)* This is because some of the dependencies your project has depend on the options you choose while quickstarting it.

You should now be able to start the newly create project with the `gearbox serve` command:

```
(tgenv)$ gearbox serve --reload
Starting subprocess with file monitor
Starting server in PID 32797.
serving on http://127.0.0.1:8080
```

Note: The `--reload` option makes the server restart whenever a file is changed, this greatly speeds up the development process by avoiding having to manually restart the server whenever we need to try our changes.

Pointing your browser to <http://127.0.0.1:8080/> should open up the TurboGears2 welcome page. By default newly quickstarted projects provide a bunch of pages to guide the user through some of the foundations of TurboGears2 web applications. Taking a look at the <http://127.0.0.1:8080/about> page can provide a great overview of your newly quickstarted project.

1.3.2 Creating and Managing Wiki Pages

If you correctly quickstarted your project with sqlalchemy database support and authentication you should end up having a `model` directory which contains database layer initialization and `User`, `Group` and `Permission` models.

Those are the standard turbogears2 authentication models. You can freely customize them, but for now we will stick to the standard ones.

To manage our pages we are going to add model that represent a Wiki Page with attributes to store the *title* of the page, page *data* and last time the page got modified.

WikiPage Model

To define the model we are going to add a `wiki.py` file inside the `wikir/model` directory which contains the model definition itself:

```
# -*- coding: utf-8 -*-
from sqlalchemy import *
from sqlalchemy.orm import mapper, relation, relation, backref
from sqlalchemy import Table, ForeignKey, Column
from sqlalchemy.types import Integer, Unicode, DateTime

from wikir.model import DeclarativeBase, metadata, DBSession
from datetime import datetime

class WikiPage(DeclarativeBase):
    __tablename__ = 'wiki_page'

    uid = Column(Integer, primary_key=True)
    updated_at = Column(DateTime, default=datetime.utcnow, nullable=False)
    title = Column(Unicode(255), nullable=False, unique=True)
    data = Column(Unicode(4096), nullable=False, default='')
```

Now to let TurboGears know that our model exists we must make it available inside the `wikir/model/__init__.py` file just by importing it at the end:

```
# Import your model modules here.
from wikir.model.auth import User, Group, Permission
from wikir.model.wiki import WikiPage
```

Creating Tables and setting up Application

Now that our model is recognized by TurboGears we must create the table that it is going to use to store its data. By default TurboGears will automatically create tables for each model it is aware of. This is performed during the application setup phase.

To setup your application you simply need to run the `gearbox setup-app` command where your application configuration file is available (usually the root of the project):

```
(tg22env)$ gearbox setup-app
Running setup_app() from wikir.websetup
Creating tables
```

The Application setup process, apart from creating tables for the known models, will also execute the `wikir/websetup/bootstrap.py` module, which by default is going to create an administrator user for our application.

Managing Pages through the Admin

Through the *manager* user that has been created during the setup phase it is possible to get access to the TurboGears Admin at <http://localhost:8080/admin>. The first time the page is accessed it will ask for authentication. Simply provide the username and password of the user that the setup-app command created for us:

```
Username: manager
Password: managepass
```

You should end up being redirected to the administration page. One of the links on the page should point to WikiPages administration page <http://localhost:8080/admin/wikipages/>

On this page a list of the existing pages is provided with a link to create a **New WikiPage**.

Note: If you don't find the WikiPages link on the administrator page, make sure you correctly imported the WikiPage model at the end of `wikir/model/__init__.py` and run the setup-app command again.

Customizing Management

Now that we have a working administration page for our WikiPages, we are going to tune a bunch of things to improve it.

First of all we are going to hide the **updated_at** fields. This will get automatically updated to the current time, so we don't really want to let users modify it.

Then if you tried to click the **New WikiPage** link you probably saw that for the title of our web page a `TextArea` is used, probably a `TextField` would be a better match to make more clear that the user is supposed to provide a short single line title.

Last but not least we are going to provide a bit more space for the page data, to make it easier to edit the page.

All these changes can be made from our model by specifying a special attribute called `__sprox__` which will be used by the administrative interface to tune the look and feel of the tables and forms it is going to generate:

```
# -*- coding: utf-8 -*-
"""Wiki Page module."""

from sqlalchemy import *
from sqlalchemy.orm import mapper, relation, relation, backref
from sqlalchemy import Table, ForeignKey, Column
```

```

from sqlalchemy.types import Integer, Unicode

from wikir.model import DeclarativeBase, metadata, DBSession
from datetime import datetime

from tw2.forms import TextField
from tw2.core import IntValidator

class WikiPage(DeclarativeBase):
    __tablename__ = 'page'

    uid = Column(Integer, primary_key=True)
    updated_at = Column(DateTime, default=datetime.utcnow, nullable=False)
    title = Column(Unicode(255), nullable=False, unique=True)
    data = Column(Unicode(4096), nullable=False, default='')

    class __sprox__(object):
        hide_fields = ['updated_at']
        field_widget_args = {'data': {'rows':15}}

```

Going back to our administration page at <http://localhost:8080/admin/wikipages/> and clicking on the **New WikiPage** link you will see a form with just a single line entry field for the title and a wide textarea for the page data.

Feel free to add as many pages as you like; we are going to see later how to display them.

1.3.3 Serving Wiki Pages

We are now able to create, edit and delete Wiki Pages, but we are still unable to serve them.

Without serving pages our wiki is actually useless, so we are going to add a controller and template to make them available.

WebSite Index

To make our wiki navigable we are going to create a new index page with a sidebar containing all the available wiki pages, so the user can easily move around.

Pages Slug and Content

To create links to the pages and display their content we are going to add `url` and `html_content` properties to the page model. The first property will create the slug for the model and provide the url where the page is available, while the second will give back the page content parsed accordingly to the [Markdown](#) language.

To generate the slugs we are going to use `tgext.datahelpers`, so the first thing we are going to do is add it to our project `setup.py` file inside the `install_requires` list:

```

install_requires=[
    "TurboGears2 >= 2.3.4",
    "Genshi",
    "zope.sqlalchemy >= 0.4",
    "sqlalchemy",
    "alembic",
    "repoze.who",
    "tw2.forms",
    "tgext.admin >= 0.6.1",

```

```
"webhelpers",
"tgext.datahelpers"
]
```

Then we need to run again `pip install -e .` to install our new project dependency:

```
(tg22env)$ pip install -e .
Successfully installed tgext.datahelpers wikir
Cleaning up...
```

Now that we installed the datahelpers we can add the `url` and `html_content` properties to our WikiPage model. Our model should end up looking like:

```
#all the other sqlalchemy imports here...
import tg
from tgext.datahelpers.utils import slugify
from webhelpers.html.converters import markdown

class WikiPage(DeclarativeBase):
    __tablename__ = 'page'

    uid = Column(Integer, primary_key=True)
    updated_at = Column(DateTime, default=datetime.utcnow, nullable=False)
    title = Column(Unicode(255), nullable=False, unique=True)
    data = Column(Unicode(4096), nullable=False, default='')

    @property
    def url(self):
        return tg.url('//'+slugify(self, self.title))

    @property
    def html_content(self):
        return markdown(self.data)

    class __sprox__(object):
        hide_fields = ['updated_at']
        field_widget_args = {'data': {'rows':15}}
```

Index Controller

Now that we are able to retrieve the url for each wiki page, we need to retrieve the list of the wiki pages with their urls so that our index page can display the sidebar.

Our index page is a wiki page itself, so we are also going to load up it's content from the page titled "index".

To do so we must edit the `RootController` class inside the `wikir/controllers/root.py` file and look for the `index` method. When you found it change it to look like:

```
@expose('wikir.templates.index')
def index(self):
    wikipages = [(w.url, w.title) for w in DBSession.query(model.WikiPage).filter(model.WikiPage.title != 'index')]

    indexpage = DBSession.query(model.WikiPage).filter_by(title='index').first()
    if not indexpage:
        content = 'Index page not available, please create a page titled index'
    else:
        content = indexpage.html_content
```

```
return dict(page='index', wikipages=wikipages, content=content)
```

TurboGears2 controllers are just plain python methods with an `@expose` decorator. The `@expose` decorator tells to TurboGears2 which template the controller is going to display and make so that all the data that our controller returns will be available inside the template itself.

If you are still asking yourself why connecting to `http://localhost:8080/` you ended up being served by the **RootController.index** method you probably want to take a look at TurboGears2 documentation about *Writing Controllers* and try to understand how *Object Dispatch* routing works.

Index Template

Now, if you reloaded to your index page you probably already noticed that nothing changed. This is because our controller retrieved the wiki pages, but we didn't expose them in the index template in any place.

The index template is available as `wikir/templates/index.html` which is exactly the same path written inside the `@expose` decorator but with `/` replaced by dots and without the template extension.

We are going to provide a really simple template, so what is currently available inside the file is going to just be removed and replaced with:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:py="http://genshi.edgewall.org/"
      xmlns:xi="http://www.w3.org/2001/XInclude">

  <xi:include href="master.html" />

<head>
  <title>TurboGears2 Wikier Index</title>
</head>

<body>
  <div class="row">
    <div class="col-md-3">
      <ul>
        <li py:for="url, title in wikipages">
          <a href="{url}">{title}</a>
        </li>
      </ul>
    </div>
    <div class="col-md-9">
      <div>
        {Markup(content)}
      </div>
    </div>
  </div>
</body>
</html>
```

Serving all Wiki pages

If you tried clicking on any link in our sidebar you probably noticed that they all lead to a 404 page. This is because we still haven't implemented any controller method that is able to serve them.

Page Template

First we are going to create a template for our wiki pages and save it as `wikir/templates/page.html`. The content of our template will look like:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:py="http://genshi.edgewall.org/"
      xmlns:xi="http://www.w3.org/2001/XInclude">

    <xi:include href="master.html" />

<head>
    <title>${title}</title>
</head>

<body>
    <div class="row">
        <div class="col-md-12">
            <h2>${title}</h2>
            ${Markup(content)}
            <a py:if="request.identity and 'managers' in request.identity['groups']"
                href="${tg.url('/admin/wikipages/%s/edit' % page_id)}">
                edit
            </a>
        </div>
    </div>
</body>
</html>
```

Page Controller

Now that we have our template we just need to bind it a controller which is going to render the page. To do this we are going to use the special `_default` controller method. This is a method that turbogears will call if it's unable to find the exact method request by the url.

As our wiki pages have a all different names they will all end up in `_default` and we will be able to serve them from there. Just edit `wikir/controller/root.py` and add the `_default` method to the `RootController`:

```
from tg import validate
from ttext.datahelpers.validators import SQLAlchemyConverter
from ttext.datahelpers.utils import fail_with

@expose('wikir.templates.page')
@validate({'page': SQLAlchemyConverter(model.WikiPage, slugified=True)},
         error_handler=fail_with(404))
def _default(self, page, *args, **kw):
    return dict(page_id=page.uid, title=page.title, content=page.html_content)
```

The `@validate` decorator makes possible to apply validators to the incoming parameters and if validation fails the specified `error_handler` is called. In this case we are checking if there is a web page with the given slug. If it fails to find one it will just return a 404 page.

If the page is available the page instance is returned, so our controller ends just returning the data of the page to the template.

If you now point your browser to the index and click any of the links in the sidebar you will see that they now lead to the linked page instead of failing with a 404 like before.

Note: If you don't have any links in the left bar, just go to the admin page and create as many pages as you like.

Our wiki is actually finished, but in the upcoming sections we are going to see how we can improve it by introducing caching.

1.3.4 Advanced Admin Customizations

TurboGears admin configurations work through the `TGAdminConfig` class, which makes it possible to change the behavior for each model. We are going to use the `EasyCrudRestController` to perform quick tuning of our administrative interface.

Displaying the Slug

Right now our admin shows us the page id and title, but doesn't provide a link to the page itself, so it's hard to see how a page looks after we edit it.

To solve this issue we are going to replace the page id with a link to the page itself inside the administration table.

Custom Admin Config

The first step is provide a custom admin config which removes the page id field. We are going to add this in `wikir/controllers/root.py`:

```
from tgext.crud import EasyCrudRestController
from tgext.admin.config import CrudRestControllerConfig

class WikiPageAdminController(EasyCrudRestController):
    __table_options__ = {'__omit_fields__': ['uid']}

class CustomAdminConfig(TGAdminConfig):
    class wiki(CrudRestControllerConfig):
        defaultCrudRestController = WikiPageAdminController
```

Once you declared your custom admin config, inside your `RootController` there should be a line which looks like:

```
admin = AdminController(model, DBSession, config_type=TGAdminConfig)
```

Replace that one with:

```
admin = AdminController(model, DBSession, config_type=CustomAdminConfig)
```

When you reload the wiki pages administration table you should see that the page uid is not there anymore.

The Slug Column

We are now going to replace the previous `uid` field with a `url` column which contains a link to the page.

To do so we have to tell our table that there an html type column (so its content doesn't get escaped) and how to generate the content for that column. This can be done inside the `WikiPageAdminController` that we just declared:

```
class WikiPageAdminController(EasyCrudRestController):
    __table_options__ = {
        '__omit_fields__': ['uid'],
        '__field_order__': ['url'],
        '__xml_fields__': ['url'],

        'url': lambda filler, row: '<a href="%s">%s</a>' % dict(url=row.url)
    }
```

Note: The `__field_order__` option is necessary to let the admin know that we have a `url` field that we want to show. Otherwise it will just know how to show it thanks to the `__xml_fields__` and `slug` properties but won't know where it has to be displayed.

Extending the Admin Further

If you want to further customize the admin behaviour have a look at the *Working with the TurboGears Admin* documentation.

1.3.5 Wiki Page Generation Caching

This section of the tutorial will show how to use the `updated_at` field of our models for different kinds of caching that can greatly speed up our website.

Updating `updated_at`

Right now the `updated_at` field of our models will only contain the time they get created, because we are never updating it. We could add a SQLAlchemy event that updates it each time the `WikiPage` object is updated, and that would probably be the suggested way to handle this.

As that solution would be out of scope for this tutorial, we are going to perform the same by customizing the TurboGears Admin.

To do so we have to customize the `WikiPageAdminController.put` method:

```
from tgext.admin.config import CrudRestControllerConfig
from datetime import datetime

class WikiPageAdminController(EasyCrudRestController):
    __table_options__ = {'__omit_fields__': ['uid'],
                        '__field_order__': ['url'],
                        '__xml_fields__': ['url'],

                        'url': lambda filler, row: '<a href="%s">%s</a>' % dict(url=row.url)
    }

    @expose(inherit=True)
    def put(self, *args, **kw):
        kw['updated_at'] = datetime.utcnow()
        return super(WikiPageAdminController, self).put(*args, **kw)
```

This way each time a wiki page is modified its `updated_at` field will be updated accordingly.

Caching Page

Now that we know when the page got updated we can use it speed up our wiki by caching wiki page generation.

The first thing we need to do is move the html content generation inside our template instead of using it directly from the controller. This can easily be done by updating our `RootController._default` method accordingly:

```
@expose('wikir.templates.page')
@validate({'page':SQLAEntityConverter(model.WikiPage, slugified=True)},
         error_handler=fail_with(404))
def _default(self, page, *args, **kw):
    return dict(wikipage=page)
```

Our controller now just retrieves the page and passes it to our template, so we have to do some minor tuning to the `wikir/templates/page.html` template too:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:py="http://genshi.edgewall.org/"
      xmlns:xi="http://www.w3.org/2001/XInclude">

    <xi:include href="master.html" />

<head>
    <title>${wikipage.title}</title>
</head>

<body>
    <div class="row">
        <div class="col-md-12">
            <h2>${wikipage.title}</h2>
            ${Markup(wikipage.html_content)}
            <a py:if="request.identity and 'managers' in request.identity['groups']"
              href="${tg.url('/admin/wikipages/%s/edit' % wikipage.uid)}">
                edit
            </a>
        </div>
    </div>
</body>
</html>
```

Now that the work of converting the page markdown content to HTML is done by our template we can simply cache the template rendering process.

This way we will both skip the template generation phase and the page content conversion phase at once. TurboGears2 provides a great tool for template caching. You just need to generate a cache key and provide it inside the `tg_cache` dictionary returned by your controller:

```
@expose('wikir.templates.page')
@validate({'page':SQLAEntityConverter(model.WikiPage, slugified=True)},
         error_handler=fail_with(404))
def _default(self, page, *args, **kw):
    cache_key = '%s-%s' % (page.uid, page.updated_at.strftime('%Y%m%d%H%M%S'))
    return dict(wikipage=page, tg_cache={'key':cache_key, 'expire':24*3600, 'type':'memory'})
```

This will keep our template cached in memory up to a day and will still regenerate the page whenever our wikipage changes as we are using the `updated_at` field to generate our cache key.

Page Caching Performances Gain

By just the minor change of caching the template the throughput of the applications on my computer greatly increased. A quick benchmark can give the idea of the impact of such a change:

```
$ /usr/sbin/ab -c 1 -n 500 http://127.0.0.1:8080/this-is-my-first-page-1
Requests per second:    97.55 [#/sec] (mean)
```

```
$ /usr/sbin/ab -c 1 -n 500 http://127.0.0.1:8080/this-is-my-first-page-1
Requests per second:    267.18 [#/sec] (mean)
```

Basic Documentation

2.1 Writing Controllers

The nerve center of your TurboGears application is **the controller**. It ultimately handles all user actions, because every HTTP request arrives here first. The controller acts on the request and can call upon other TurboGears components (the template engines, database layers, etc.) as its logic directs.

2.1.1 Basic Dispatch

When the TurboGears server receives an HTTP request, the requested URL is mapped as a call to your controller code located in the `controllers` package. Page names map to other controllers or methods within the controller class.

For example:

URL	Maps to
<code>http://localhost:8080/index</code>	<code>RootController.index()</code>
<code>http://localhost:8080/mypage</code>	<code>RootController.mypage()</code>

2.1.2 Index and Catch-All pages

Suppose using `gearbox quickstart` you generate a TurboGears project named “HelloWorld”. Your default controller code would be created in the file `HelloWorld/helloworld/controllers/root.py`.

Modify the default `root.py` to read as follows:

```
"""Main Controller"""
from helloworld.lib.base import BaseController
from tg import expose, flash
#from tg import redirect, validate
#from helloworld.model import DBSession

class RootController(BaseController):
    @expose()
    def index(self):
        return "<h1>Hello World</h1>"

    @expose()
    def _default(self, *args, **kw):
        return "This page is not ready"
```

When you load the root URL `http://localhost:8080/index` in your web browser, you'll see a page with the message "Hello World" on it. In addition, any of [these URLs](#) will return the same result.

Implementing A Catch-All Url Via The `_default ()` Method

URLs not explicitly mapped to other methods of the controller will generally be directed to the method named `_default()`. With the above example, requesting any URL besides `/index`, for example `http://localhost:8080/hello`, will return the message "This page is not ready".

Adding More Pages

When you are ready to add another page to your site, for example at the URL

```
http://localhost:8080/anotherpage
```

add another method to class `RootController` as follows:

```
@expose()
def anotherpage(self):
    return "<h1>There are more pages in my website</h1>"
```

Now, the URL `/anotherpage` will return:

There are more pages in my website

Line By Line Explanation

```
"""Main Controller"""
from helloworld.lib.base import BaseController
from tg import expose, flash
from tg.i18n import ugettext as _
#from tg import redirect, validate
#from helloworld.model import DBSession
```

First you need to import the required modules.

There's a lot going on here, including some stuff for internationalization. But we're going to gloss over some of that for now. The key thing to notice is that you are importing a `BaseController`, which your `RootController` must inherit from. If you're particularly astute, you'll have noticed that you import this `BaseController` from the `lib` module of your own project, and not from TurboGears.

TurboGears provides *ObjectDispatch* system through the `TGController` class which is imported in the `lib` folder of the current project (`HelloWorld/helloworld/lib`) so that you can modify it to suit the needs of your application. For example, you can define actions which will happen on every request, add parameters to every template call, and otherwise do what you need to the request on the way in, and on the way out.

The next thing to notice is that we are importing `expose` from `tg`.

`BaseController` classes and the `expose` decorator are the basis of TurboGears controllers. The `@expose` decorator declares that your method should be *exposed to the web*, and provides you with the ability to say how the results of the controller should be rendered.

The other imports are there in case you do internationalization, use the HTTP redirect function, validate inputs/outputs, or use the models.

```
class RootController(BaseController):
```

`RootController` is the required standard name for the `RootController` class of a TurboGears application and it should inherit from the `BaseController` class. It is thereby specified as the request handler class for the website's root.

In TurboGears 2 the web site is represented by a tree of controller objects and their methods, and a TurboGears website always grows out from the `RootController` class.

```
def index(self):
    return "<h1>Hello World</h1>"
```

We'll look at the methods of the `RootController` class next.

The `index` method is the start point of any TurboGears controller class. Each of the URLs

- `http://localhost:8080`
- `http://localhost:8080/`
- `http://localhost:8080/index`

is mapped to the `RootController.index()` method.

If a URL is requested and does not map to a specific method, the `_default()` method of the controller class is called:

```
def _default(self):
    return "This page is not ready"
```

In this example, all pages except the [these urls](#) listed above will map to the `_default` method.

As you can see from the examples, the response to a given URL is determined by the method it maps to.

```
@expose()
```

The `@expose()` seen before each controller method directs TurboGears controllers to make the method accessible through the web server. Methods in the controller class that are *not* "exposed" can not be called directly by requesting a URL from the server.

There is much more to `@expose()`. It will be our access to TurboGears sophisticated rendering features that we will explore shortly.

2.1.3 Exposing Templates

As shown above, controller methods return the data of your website. So far, we have returned this data as literal strings. You could produce a whole site by returning only strings containing raw HTML from your controller methods, but it would be difficult to maintain, since Python code and HTML code would not be cleanly separated.

Expose + Template == Good

To enable a cleaner solution, data from your TurboGears controller can be returned as strings, **or** as a dictionary.

With `@expose()`, a dictionary can be passed from the controller to a template which fills in its placeholder keys with the dictionary values and then returns the filled template output to the browser.

Template Example

A simple template file called `sample` could be made like this:

```
<html>
  <head>
<title>TurboGears Templating Example</title>
  </head>
  <body>
    <h2>I just want to say that ${person} should be the next
      ${office} of the United States.</h2>
  </body>
</html>
```

The `${param}` syntax in the template indicates some undetermined values to be filled.

We provide them by adding a method to the controller like this ...

```
@expose("helloworld.templates.sample")
def example(self):
    mydata = {'person': 'Tony Blair', 'office': 'President'}
    return mydata
```

... then the following is made possible:

- The web user goes to `http://localhost:8080/example`.
- The `example` method is called.
- The method `example` returns a Python dict.
- `@expose` processes the dict through the template file named `sample.html`.
- The dict values are substituted into the final web response.
- The web user sees a marked up page saying:

I just want to say that Tony Blair should be the next President of the United States.

Template files can thus house all markup information, maintaining clean separation from controller code.

For more on templating have a look at *Templating*

2.1.4 SubControllers And The URL Hierarchy

Sometimes your web-app needs a URL structure that's more than one level deep.

TurboGears provides for this by traversing the object hierarchy, to find a method that can handle your request.

To make a sub-controller, all you need to do is make your sub-controller inherit from the object class. However there's a `SubController` class `Controller` in your project's `lib.base` (`HelloWorld/helloworld/lib/base.py`) for you to use if you want a central place to add helper methods or other functionality to your SubControllers:

```
from lib.base import BaseController
from tg import redirect

class MovieController(BaseController):
    @expose()
    def index(self):
        redirect('list/')

    @expose()
    def list(self):
        return 'hello'
```

```
class RootController(BaseController):
    movie = MovieController()
```

With these in place, you can follow the link:

- <http://localhost:8080/movie/>
- <http://localhost:8080/movie/index>

and you will be redirected to:

- <http://localhost:8080/movie/list/>

Unlike turbogears 1, going to <http://localhost:8080/movie> **will not** redirect you to <http://localhost:8080/movie/list>. This is due to some interesting bit about the way WSGI works. But it's also the right thing to do from the perspective of URL joins. Because you didn't have a trailing slash, there's no way to know you meant to be in the movie directory, so redirection to relative URLs will be based on the last / in the URL. In this case the root of the site.

It's easy enough to get around this, all you have to do is write your redirect like this:

```
redirect('/movie/list/')
```

Which provides the redirect method with an absolute path, and takes you exactly where you wanted to go, no matter where you came from.

2.1.5 Passing Parameters To The Controller

Now that you have the basic routing dispatch understood, you may be wondering how parameters are passed into the controller methods. After all, a framework would not be of much use unless it could accept data streams from the user.

TurboGears uses introspection to assign values to the arguments in your controller methods. This happens using the same duck-typing you may be familiar with if you are a frequent python programmer. Here is the basic approach:

- **The dispatcher gobbles up as much of the URL as it can to find the** correct controller method associated with your request.
- The remaining url items are then mapped to the parameters in the method.
- If there are still remaining parameters they are mapped to `*args` in the method signature.
- **If there are named parameters, (as in a form request, or a GET request with parameters), they are mapped to the** args which match their names, and if there are leftovers, they are placed in `**kw`.

Here is an example controller and a chart outlining the way urls are mapped to it's methods:

```
class WikiController(TGController):

    def index(self):
        """returns a list of wiki pages"""
        ...

    def _default(self, *args):
        """returns one wikipage"""
        ...

    def create(self, title, text, author='anonymous', **kw):
        wikipage = Page(title=title, text=text, author=author, tags=str(kw))
        DBSession.add(wikipage)

    def update(self, title, **kw):
        wikipage = DBSession.query(Page).get(title)
```

```

for key, value in kw:
    setattr(wikipage, key, value)

def delete(self, title):
    wikipage = DBSession.query(Page).get(title)
    DBSession.delete(wikipage)

```

URL	Method	Argument Assignments
/	index	
/NewPage	_default	args : ['NewPage'] text: 'More Information'
/create/NewPage?text=More Information	create title: 'NewPage'	kw: {'author':'Lenny'}
/update/NewPage?author=Lenny	update title: 'NewPage'	
/delete/NewPage	delete	title : 'NewPage'

The parameters that are turned into arguments arrive in string format. It is a good idea to use Python's type casting capabilities to change the arguments into the types the rest of your program expects. For instance, if you pass an integer 'id' into your function you might use `id = int(id)` to cast it into an int before usage. Another way to accomplish this feat is to use the `@validate` decorator, which is explained in [TurboGears Validation](#)

Ignore Unused Parameters

By default TurboGears2 will complain about parameters that the controller method was not expecting. If this is causing any issue as you need to share between all the urls a parameter that it is used by your javascript framework or for any other reason, you can use `ignore_parameters` option to have TurboGears2 ignore them. Just add the list of parameters to ignore in `config/app_cfg.py`:

```
base_config.ignore_parameters = ['timestamp', 'param_name']
```

You will still be able to access them from the `tg.request` object if you need them for any reason.

2.2 Templating

TurboGears enables template rendering though the `tg.decorators.expose` decorator to link controller methods to template and through the `tg.render_template()` function.

Each template is rendered using a *template engine*, TurboGears provides some builtin engines but additional can be configured. The `default_renderer` for TurboGears applications is Genshi which permits to write templates in pure xhtml and validates them to detect issues at compile time and prevent serving broken pages. For documentation on Genshi templates see the [Genshi Template Language](#).

By default TurboGears references to templates using *dotted notation*, this is the path of the template file in terms of python packages. This makes possible to refer to template files independently from where the application is installed and started as it refers to the python package where the template file is provided.

Typical dotted notation path looks like: **mypackage.templates.template_file** and it doesn't include any extention. If an extension is provided TurboGears will try to read the path as a file system path, not as a dotted notation path.

2.2.1 Explicit Engine in Exposition

The `@expose` decorator template files will always be rendered using the `default_renderer` specified into the application configurator unless explicitly set. To explicitly provide the template engine to use just prepend it to the template path in the form **engine:template_path** like **genshi:mypackage.templates.template_file**.

Refer to *Rendering Engines Configuration* documentation for information on setting up available renderers and specifying the default one.

2.2.2 Template Variables

To pass template variables the controller is expected to return a `dict` with all the variables inside. Those will be available inside the template with the same name.

In addition to variables explicitly specified by the user, TurboGears adds some additional variables and utilities. The most useful one are probably:

- **h** which is the `lib.helpers` module, this usually includes every utility function for formatting text and html in templates.
- **request, response, tmpl_context, app_globals, config** which are the same available inside controllers.
- **identity** which is the currently logged user when recognized
- **tg.url** which is the utility function to create urls in TurboGears.

For a complete list of those variables refer to the `tg.render_template()` documentation. You can add additional variable to every single template by setting a `variable_provider` function inside the Application Configurator (`app_cfg.base_config` object).

This function is expected to return a `dict` with any variable that should be added to the default template variables. It can even replace existing variables.

2.3 Genshi XML Template Language

Genshi provides a XML-based template language that is heavily inspired by [Kid](#), which in turn was inspired by a number of existing template languages, namely [XSLT](#), [TAL](#), and [PHP](#).

This document describes the template language and will be most useful as reference to those developing Genshi XML templates. Templates are XML files of some kind (such as XHTML) that include processing [directives](#) (elements or attributes identified by a separate namespace) that affect how the template is rendered, and template expressions that are dynamically substituted by variable data.

2.3.1 Template Directives

Directives are elements and/or attributes in the template that are identified by the namespace `http://genshi.edgewall.org/`. They can affect how the template is rendered in a number of ways: Genshi provides directives for conditionals and looping, among others.

To use directives in a template, the namespace must be declared, which is usually done on the root element:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:py="http://genshi.edgewall.org/"
      lang="en">
  ...
</html>
```

In this example, the default namespace is set to the XHTML namespace, and the namespace for Genshi directives is bound to the prefix “py”.

All directives can be applied as attributes, and some can also be used as elements. The `if` directives for conditionals, for example, can be used in both ways:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:py="http://genshi.edgewall.org/"
      lang="en">
  ...
  <div py:if="foo">
    <p>Bar</p>
  </div>
  ...
</html>
```

This is basically equivalent to the following:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:py="http://genshi.edgewall.org/"
      lang="en">
  ...
  <py:if test="foo">
    <div>
      <p>Bar</p>
    </div>
  </py:if>
  ...
</html>
```

The rationale behind the second form is that directives do not always map naturally to elements in the template. In such cases, the `py:strip` directive can be used to strip off the unwanted element, or the directive can simply be used as an element.

Conditional Sections

`py:if`

The element and its content is only rendered if the expression evaluates to a truth value:

```
<div>
  <b py:if="foo">${bar}</b>
</div>
```

Given the data `foo=True` and `bar='Hello'` in the template context, this would produce:

```
<div>
  <b>Hello</b>
</div>
```

But setting `foo=False` would result in the following output:

```
<div>
</div>
```

This directive can also be used as an element:

```
<div>
  <py:if test="foo">
    <b>${bar}</b>
  </py:if>
</div>
```

py:choose

The `py:choose` directive, in combination with the directives `py:when` and `py:otherwise` provides advanced conditional processing for rendering one of several alternatives. The first matching `py:when` branch is rendered, or, if no `py:when` branch matches, the `py:otherwise` branch is rendered.

If the `py:choose` directive is empty the nested `py:when` directives will be tested for truth:

```
<div py:choose="">
  <span py:when="0 == 1">0</span>
  <span py:when="1 == 1">1</span>
  <span py:otherwise="">2</span>
</div>
```

This would produce the following output:

```
<div>
  <span>1</span>
</div>
```

If the `py:choose` directive contains an expression the nested `py:when` directives will be tested for equality to the parent `py:choose` value:

```
<div py:choose="1">
  <span py:when="0">0</span>
  <span py:when="1">1</span>
  <span py:otherwise="">2</span>
</div>
```

This would produce the following output:

```
<div>
  <span>1</span>
</div>
```

These directives can also be used as elements:

```
<py:choose test="1">
  <py:when test="0">0</py:when>
  <py:when test="1">1</py:when>
  <py:otherwise>2</py:otherwise>
</py:choose>
```

Looping**py:for**

The element is repeated for every item in an iterable:

```
<ul>
  <li py:for="item in items">${item}</li>
</ul>
```

Given `items=[1, 2, 3]` in the context data, this would produce:

```
<ul>
  <li>1</li><li>2</li><li>3</li>
</ul>
```

This directive can also be used as an element:

```
<ul>
  <py:for each="item in items">
    <li>${item}</li>
  </py:for>
</ul>
```

Snippet Reuse

`py:def`

The `py:def` directive can be used to create macros, i.e. snippets of template code that have a name and optionally some parameters, and that can be inserted in other places:

```
<div>
  <p py:def="greeting(name)" class="greeting">
    Hello, ${name}!
  </p>
  ${greeting('world')}
  ${greeting('everyone else')}
</div>
```

The above would be rendered to:

```
<div>
  <p class="greeting">
    Hello, world!
  </p>
  <p class="greeting">
    Hello, everyone else!
  </p>
</div>
```

If a macro doesn't require parameters, it can be defined without the parenthesis. For example:

```
<div>
  <p py:def="greeting" class="greeting">
    Hello, world!
  </p>
  ${greeting()}
</div>
```

The above would be rendered to:

```
<div>
  <p class="greeting">
    Hello, world!
  </p>
</div>
```

This directive can also be used as an element:

```
<div>
  <py:def function="greeting(name)">
    <p class="greeting">Hello, ${name}!</p>
  </py:def>
</div>
```

`py:match`

This directive defines a *match template*: given an XPath expression, it replaces any element in the template that matches the expression with its own content.

For example, the match template defined in the following template matches any element with the tag name “greeting”:

```
<div>
  <span py:match="greeting">
    Hello ${select('@name')}
  </span>
  <greeting name="Dude" />
</div>
```

This would result in the following output:

```
<div>
  <span>
    Hello Dude
  </span>
</div>
```

Inside the body of a `py:match` directive, the `select(path)` function is made available so that parts or all of the original element can be incorporated in the output of the match template. See [Using XPath](#) for more information about this function.

Match templates are applied both to the original markup as well to the generated markup. The order in which they are applied depends on the order they are declared in the template source: a match template defined after another match template is applied to the output generated by the first match template. The match templates basically form a pipeline.

This directive can also be used as an element:

```
<div>
  <py:match path="greeting">
    <span>Hello ${select('@name')}</span>
  </py:match>
  <greeting name="Dude" />
</div>
```

When used this way, the `py:match` directive can also be annotated with a couple of optimization hints. For example, the following informs the matching engine that the match should only be applied once:

```
<py:match path="body" once="true">
  <body py:attrs="select('@*')">
    <div id="header">...</div>
    ${select("*|text()")}
    <div id="footer">...</div>
  </body>
</py:match>
```

The following optimization hints are recognized:

At-tribute	De-fault	Description
<code>buffer</code>	<code>true</code>	Whether the matched content should be buffered in memory. Buffering can improve performance a bit at the cost of needing more memory during rendering. Buffering is “required” for match templates that contain more than one invocation of the <code>select()</code> function. If there is only one call, and the matched content can potentially be very long, consider disabling buffering to avoid excessive memory use.
<code>once</code>	<code>false</code>	Whether the engine should stop looking for more matching elements after the first match. Use this on match templates that match elements that can only occur once in the stream, such as the <code><head></code> or <code><body></code> elements in an HTML template, or elements with a specific ID.
<code>recursive</code>	<code>true</code>	Whether the match template should be applied to its own output. Note that <code>once</code> implies non-recursive behavior, so this attribute only needs to be set for match templates that don’t also have <code>once</code> set.

Note: The `py:match` optimization hints were added in the 0.5 release. In earlier versions, the attributes have no effect.

Variable Binding

`py:with`

The `py:with` directive lets you assign expressions to variables, which can be used to make expressions inside the directive less verbose and more efficient. For example, if you need use the expression `author.posts` more than once, and that actually results in a database query, assigning the results to a variable using this directive would probably help.

For example:

```
<div>
  <span py:with="y=7; z=x+10">$x $y $z</span>
</div>
```

Given `x=42` in the context data, this would produce:

```
<div>
  <span>42 7 52</span>
</div>
```

This directive can also be used as an element:

```
<div>
  <py:with vars="y=7; z=x+10">$x $y $z</py:with>
</div>
```

Note that if a variable of the same name already existed outside of the scope of the `py:with` directive, it will **not** be overwritten. Instead, it will have the same value it had prior to the `py:with` assignment. Effectively, this means that variables are immutable in Genshi.

Structure Manipulation

`py:attrs`

This directive adds, modifies or removes attributes from the element:

```
<ul>
  <li py:attrs="foo">Bar</li>
</ul>
```

Given `foo={'class': 'collapse'}` in the template context, this would produce:

```
<ul>
  <li class="collapse">Bar</li>
</ul>
```

Attributes with the value `None` are omitted, so given `foo={'class': None}` in the context for the same template this would produce:

```
<ul>
  <li>Bar</li>
</ul>
```

This directive can only be used as an attribute.

py:content

This directive replaces any nested content with the result of evaluating the expression:

```
<ul>
  <li py:content="bar">Hello</li>
</ul>
```

Given `bar='Bye'` in the context data, this would produce:

```
<ul>
  <li>Bye</li>
</ul>
```

This directive can only be used as an attribute.

py:replace

This directive replaces the element itself with the result of evaluating the expression:

```
<div>
  <span py:replace="bar">Hello</span>
</div>
```

Given `bar='Bye'` in the context data, this would produce:

```
<div>
  Bye
</div>
```

This directive can also be used as an element (since version 0.5):

```
<div>
  <py:replace value="title">Placeholder</py:replace>
</div>
```

`py:strip`

This directive conditionally strips the top-level element from the output. When the value of the `py:strip` attribute evaluates to `True`, the element is stripped from the output:

```
<div>
  <div py:strip="True"><b>foo</b></div>
</div>
```

This would be rendered as:

```
<div>
  <b>foo</b>
</div>
```

As a shorthand, if the value of the `py:strip` attribute is empty, that has the same effect as using a truth value (i.e. the element is stripped).

Processing Order

It is possible to attach multiple directives to a single element, although not all combinations make sense. When multiple directives are encountered, they are processed in the following order:

1. `py:def`
2. `py:match`
3. `py:when`
4. `py:otherwise`
5. `py:for`
6. `py:if`
7. `py:choose`
8. `py:with`
9. `py:replace`
10. `py:content`
11. `py:attrs`
12. `py:strip`

2.3.2 Includes

To reuse common snippets of template code, you can include other files using `XInclude`.

For this, you need to declare the `XInclude` namespace (commonly bound to the prefix “`xi`”) and use the `<xi:include>` element where you want the external file to be pulled in:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:py="http://genshi.edgewall.org/"
      xmlns:xi="http://www.w3.org/2001/XInclude">
  <xi:include href="base.html" />
  ...
</html>
```

Include paths are relative to the filename of the template currently being processed. So if the example above was in the file “myapp/index.html” (relative to the template search path), the XInclude processor would look for the included file at “myapp/base.html”. You can also use Unix-style relative paths, for example “. ./base.html” to look in the parent directory.

Any content included this way is inserted into the generated output instead of the `<xi:include>` element. The included template sees the same context data. [Match templates](#) and [macros](#) in the included template are also available to the including template after the point it was included.

By default, an error will be raised if an included file is not found. If that’s not what you want, you can specify fallback content that should be used if the include fails. For example, to make the include above fail silently, you’d write:

```
<xi:include href="base.html"><xi:fallback /></xi:include>
```

See the [XInclude specification](#) for more about fallback content. Note though that Genshi currently only supports a small subset of XInclude.

Dynamic Includes

Includes in Genshi are fully dynamic: Just like normal attributes, the `href` attribute accepts expressions, and [directives](#) can be used on the `<xi:include />` element just as on any other element, meaning you can do things like conditional includes:

```
<xi:include href="{name}.html" py:if="not in_popup"
           py:for="name in ('foo', 'bar', 'baz')" />
```

Including Text Templates

The `parse` attribute of the `<xi:include>` element can be used to specify whether the included template is an XML template or a text template (using the new syntax added in Genshi 0.5):

```
<xi:include href="myscript.js" parse="text" />
```

This example would load the `myscript.js` file as a `NewTextTemplate`. See [text templates](#) for details on the syntax of text templates.

2.3.3 Comments

Normal XML/HTML comment syntax can be used in templates:

```
<!-- this is a comment -->
```

However, such comments get passed through the processing pipeline and are by default included in the final output. If that’s not desired, prefix the comment text with an exclamation mark:

```
<!-- !this is a comment too, but one that will be stripped from the output -->
```

Note that it does not matter whether there’s whitespace before or after the exclamation mark, so the above could also be written as follows:

```
<!--! this is a comment too, but one that will be stripped from the output -->
```

2.4 Scaffolding

Scaffolding is the process of creating a new component of your web application through a template or preset.

2.4.1 Creating new Controllers, Models and Templates

On TurboGears this is provided by the `gearbox scaffold` command which will look for `.template` files inside your project directory to create new scaffolds for your web app.

Since version 2.3.5 projects quickstarted by TurboGears provide `.template` files for *model*, *controller* and *template* so new models, controllers and templates can be easily created in your project by running `gearbox scaffold`.

For example to create a new **Photo** model simply run:

```
$ gearbox scaffold model photo
```

It will create a `model/photo.py` file with a `Photo` class inside which you just need to import inside `model/__init__.py` to make it available inside your app.

2.4.2 Creating All Together

Multiple scaffolds can be created together, so in case you need to create a new model with a controller to handle it and an index page you can run:

```
$ gearbox scaffold model controller template photo
```

Which will create a new controller with the associated page and model. To start using the controller mount it inside your application `RootController`.

2.4.3 Creating Packages

There are cases when your controllers, templates and model might become complex and be better managed by a python package instead of a simple module. This is common for templates which will usually be more than one for each controller and so are usually grouped in a package for each controller.

To create scaffold in a package just provide the `-s [PACKAGE]` option to the scaffold command:

```
$ gearbox scaffold -s photo controller template photo
```

This will create a photo controller and template inside a photo package where multiple templates can be placed.

2.5 TurboGears Validation

When using TurboGears, your controller methods get their arguments built from the various GET, POST, and URL mechanisms provided by TurboGears. The only downside is that all the arguments will be strings and you'd like them converted to their normal Python datatype: numbers to `int`, dates to `datetime`, etc.

This conversion functionality is provided by the `FormEncode` package and is applied to your methods using the `@validate()` decorator. `FormEncode` provides both validation and conversion as a single step, reasoning that you frequently need to validate something before you can convert it or that you'll need to convert something before you can really validate it.

The `@validate()` decorator can evaluate both widget-based forms and the standard form arguments so they are not dependent on widgets at all.

Furthermore, the `@validate()` decorator is not really required at all. It just provides a convenience so that you can assume that you have the right kind of data inside your controller methods. This helps separate validation logic from application logic about what to do with valid data.

If you don't put a `@validate()` decorator on your method, you'll simply have to do the string conversion in your controller.

2.5.1 Validating Arguments

When not using forms, the story gets a bit more complex. Basically, you need to specify which validator goes with which argument using the `validators` keyword argument. Here's a simple example:

```
from formencode import validators

@expose('json')
@validate(validators={"a":validators.Int(), "b":validators.Email})
def two_validators(self, a=None, b=None, *args):
    validation_status = tg.request.validation
    errors = [{key, value} in validation_status['errors'].iteritems()]
    values = validation_status['values']
    return dict(a=a, b=b, errors=str(errors), values=str(values))
```

The dictionary passed to `validators` maps the incoming field names to the appropriate `FormEncode` validators, `Int` in this example.

If there's a validation error, TurboGears calls the `error_handler` if it exists, but it always adds `errors` and `values` to `request.validation`, so they will be available there for the rest of the request. In this case if there are validation errors, we grab both the error messages and the original *unvalidated* values and return them in the error message.

`FormEncode` provides a number of useful pre-made validators for you to use: they are available in the `formencode.validators` module.

For most validators, you can pass keyword arguments for more specific constraints.

2.5.2 Validation Process Information

TurboGears provides some information on the currently running validation process while it is handling the validation error.

Whenever an error handling is in process some properties are available in the `tg.request.validation` to provide overview of the validation error:

- `tg.request.validation['values']` The submitted values before validation
- `tg.request.validation['errors']` The errors that triggered the error handling
- `tg.request.validation['exception']` The validation exception that triggered the error handling
- `tg.request.validation['error_handler']` The error handler that is being executed

FormEncode Validators

- Attribute
- Bool

- CIDR
- ConfirmType
- Constant
- CreditCardExpires
- CreditCardSecurityCode
- CreditCardValidator
- DateConverter
- DateTime
- DateValidator
- DictConverter
- Email
- Empty
- False
- FancyValidator
- FieldStorageUploadConverter
- FieldsMatch
- FileUploadKeeper
- FormValidator
- IDeclarative
- IPhoneNumberValidator
- ISchema
- IValidator
- Identity
- IndexListConverter
- Int
- Interface
- Invalid
- MACAddress
- MaxLength
- MinLength
- NoDefault
- NotEmpty
- Number
- OneOf
- PhoneNumber
- PlainText

- PostalCode
- Regex
- RequireIfMissing
- RequireIfPresent
- Set
- SignedString
- StateProvince
- String
- StringBool
- StringBoolean
- StripField
- TimeConverter
- True
- URL
- UnicodeString
- Validator
- Wrapper

For the absolute most up-to date list of available validators, check the [FormEncode validators](#) module. You can also create your own validators or build on existing validators by inheriting from one of the defaults.

See the FormEncode documentation for how this is done.

You can also compose compound validators with logical operations, the FormEncode compound module provides *All* (all must pass), *Any* (any one must pass) and *Pipe* (all must pass with the results of each validator passed to the next item in the Pipe). You can use these like so:

```
from formencode.compound import All
...
the_validator=All(
    validators.NotEmpty(),
    validators.UnicodeString(),
)
```

2.5.3 Writing Custom Validators

If you can't or don't want to rely on the FormEncode library you can write your own validators.

Validators are simply objects that provide a `to_python` method which returns the converted value or raise `tg.validation.TGValidationError`

For example a validator that converts a paramter to an integer would look like:

```
from tg.validation import TGValidationError

class IntValidator(object):
    def to_python(self, value, state=None):
        try:
            return int(value)
```

```
except:
    raise TGValidationError('Integer expected')
```

Then it is possible to pass an instance of IntValidator to the TurboGears @validate decorator.

2.5.4 Validating Widget Based Forms

The simplest way to use @validate() is to pass in a reference to a widgets-based form:

```
@validate(projectname.forms.a_form)
```

The widgets system will take care of building a schema to handle the data conversions and you'll wind up with the int or datetime objects you specified when building the form. When paired with the validate decorator, you can handle the common case of building a form, validating it, redisplaying the form if there are errors, and converting a valid form into the proper arguments in only a few lines of Python.

You can also pass the form using a keyword argument:

```
@validate(form=projectname.forms.a_form)
```

You might also want to tell TurboGears to pass off handling of invalid data to a different controller. To do that you just pass the method you want called to @validate via the error_handler param:

```
@validate(forms.myform, error_handler=process_form_errors)
```

The method in question will be called, with the unvalidated data as its parameters. And error validation messages will be stored in tg.request.validation.

Here's a quick example of how this all works:

```
@expose('json')
@validate(form=myform)
def process_form_errors(self, **kwargs):
    #add error messages to the kwargs dictionary and return it
    kwargs['errors'] = tg.request.validation['errors']
    return dict(kwargs)

@expose('json')
@validate(form=myform, error_handler=process_form_errors)
def send_to_error_handler(self, **kwargs):
    return dict(kwargs)
```

If there's a validation error in myform, the send_to_error_handler method will never get called. Instead process_form_errors will get called, and the validation error messages can be picked up from the errors value of the request validation context.

2.5.5 Schema Validation

Sometimes you need more power and flexibility than you can get from validating individual form fields. Fortunately FormEncode provides just the thing for us – Schema validators.

If you want to do multiple-field validation, reuse validators or just clean up your code, validation Schema's are the way to go. You create a validation schema by inheriting from formencode.schema.Schema and pass the newly created Schema as the validators argument instead of passing a dictionary.

Create a schema:

```
class PwdSchema(schema.Schema):
    pwd1 = validators.String(not_empty=True)
    pwd2 = validators.String(not_empty=True)
    chained_validators = [validators.FieldsMatch('pwd1', 'pwd2')]
```

Then you can use that schema in `@validate` rather than a dictionary of validators:

```
@expose()
@validate(validators=PwdSchema())
def password(self, pwd1, pwd2):
    if tg.request.validation['errors']:
        return "There was an error"
    else:
        return "Password ok!"
```

Besides noticing our brilliant security strategy, please notice the `chained_validators` part of the schema that guarantees a pair of matching fields.

Again, for information about `Invalid` exception objects, creating your own validators, schema and `FormEncode` in general, refer to the [FormEncode Validator](#) documentation and don't be afraid to check the `Formencode.validators` source. It's often clearer than the documentation.

Note that Schema validation is rigorous by default, in particular, you must declare *every* field you are going to pass into your controller or you will get validation errors. To avoid this, add:

```
class MySchema( schema.Schema ):
    allow_extra_fields=True
```

to your schema declaration.

2.6 Displaying Flash/Notice Messages

TurboGears provides a way to display short messages inside the current or next page. This works by using the `WebFlash` module which stores short text messages inside a cookie so that it can be retrieved when needed.

2.6.1 Default Setup

By Default the `master.html` of a quickstarted project provides a div where flash messages will be displayed, this is achieved with the following lines of code:

```
<py:with vars="flash=tg.flash_obj.render('flash', use_js=False)">
    <div py:if="flash" py:replace="Markup(flash)" />
</py:with>
```

The `tg.flash_obj` is the `WebFlash` object which is available inside any rendered template. This object permits to retrieve the current flash message and display it.

2.6.2 Storing Flash Messages

Flash messages can be stored using the `tg.flash` command this allows to store a message with a status option to configure the flash style.

```
tg.flash('Message', 'status')
```

If the method that called flash performs a redirect the flash will be visible inside the redirected page. If the method directly exposes a template the flash will be visible inside the template itself.

2.6.3 Caching with Flash Messages

When using `tg_cache` variable in rendered templates (*Prerendered Templates Caches*) the flash will get into the cached template causing unwanted messages to be displayed.

To solve this issue the `tg.flash_obj.render` method provides the `use_js` option. By default this option is set at `False` inside the template, changing it to `True` will make the flash message to be rendered using javascript. This makes so that the same template is always rendered with a javascript to fetch the flash message and display it due to the fact that the template won't change anymore it will now be possible to correctly cache it.

2.6.4 Customizing Flash

CSS Styling

By default *warning*, *error*, *info*, *ok* statuses provide a style in `public/css/style.css` for quickstarted applications.

Any number of statuses can be configured using plain css:

```
#flash > .warning {
  color: #c09853;
  background-color: #fcf8e3;
  border-color: #fbed5;
}

#flash > .ok {
  color: #468847;
  background-color: #dff0d8;
  border-color: #d6e9c6;
}

#flash > .error {
  color: #b94a48;
  background-color: #f2dede;
  border-color: #eed3d7;
}

#flash > .info {
  color: #3a87ad;
  background-color: #d9edf7;
  border-color: #bce8f1;
}
```

Flash Options

Flash messages support can be styled using options inside the `flash.` namespace, those are documented in `TGFlash` and can be specified in `config/app_cfg.py` or in your `.AppConfig` instance.

For example to change the default message status (when status is omitted) you can use the `flash.default_status` option and set it to any string. To change the default flash template you can use `flash.template` and set it to a string with the HTML that should be displayed to show the flash (note that `flash.template` only works for static rendered flash, not for JS version).

Custom Flash HTML

For example to render the flash using the **toastr** library you might want to remove the `py:with` code block from your `master.html` and move it to the bottom of your `<body>` right after the usage of bootstrap and jquery libraries:

```
<body>
  <!-- YOUR CURRENT BODY CONTENT -->
  <script src="http://code.jquery.com/jquery.js"></script>
  <script src="{tg.url('/javascript/bootstrap.min.js')}"></script>

  <py:with vars="flash=tg.flash_obj.render('flash')">
    <py:if test="flash">${Markup(flash)}</py:if>
  </py:with>
</body>
```

This will ensure that we can provide custom Javascript that depends on JQuery inside our flash template. Now we can switch flash template to use the toastr library to display our flash by setting inside your `app_cfg.py`:

```
base_config['flash.default_status'] = 'success'
base_config['flash.template'] = '''\
  <script src="//cdnjs.cloudflare.com/ajax/libs/toastr.js/latest/js/toastr.min.js"></script>
  <script>toastr.$status("${message}");</script>
'''
```

This will ensure that each time the flash is displayed the toastr library with the given status is used.

Last, to correctly display the flash with the right look and feel, don't forget to add the toastr CSS to the head of your `master.html`:

```
<link rel="stylesheet" type="text/css" media="screen"
      href="//cdnjs.cloudflare.com/ajax/libs/toastr.js/latest/css/toastr.min.css" />
```

If everything is correct you should see your flash messages as balloon into the top-right corner of your webpage.

Custom Flash JavaScript

Javascript based flashes are usually common when Caching is involved, so the cached version of the webpage will not have the flash inside but you still want to be able to display the flash messages. In this case instead of providing a custom `flash.template` you should provide a custom `flash.js_call` which is the javascript code used to display the message.

For example to use the toastr library you might want to ensure toastr CSS and JS are available and add the following to your `app_cfg.py`:

```
base_config['flash.default_status'] = 'success'
base_config['flash.js_call'] = '''\
  var payload = webflash.payload();
  if(payload) { toastr[payload.status](payload.message); }
'''
```

The `webflash` object is provided by `TGFlash` itself and the `webflash.payload()` method will fetch the current message for you.

2.7 Authorization in TurboGears

This document describes how authentication is integrated into TurboGears and how you may get started with it.

2.7.1 How authentication and authorization is set up by default

If you enabled authentication and authorization in your project when it was generated by TurboGears, then it's been set up to store your users, groups and permissions in SQLAlchemy-managed tables or Ming collections.

Your users' table is used by `repoze.who` to authenticate them. When the authentication has success TurboGears uses the `TGAuthMetadata` instance declared in `config.base_config.sa_auth.authmetadata` to retrieve the informations about your user which are required for authorization.

You are free to change the `authmetadata` object as you wish, usually if your authentication model changes, your `authmetadata` object will change accordingly.

You can even get rid of authorization based on groups and permissions and use other authorization patterns (e.g., roles, based on network components) or simply use a mix of patterns. To do this you can set `authmetadata` to `None` and register your own metadata providers for `repoze.who`.

2.7.2 Restricting access with `tg.predicates`

`tg.predicates` allows you to define access rules based on so-called “predicate checkers”. It is a customized version of the `repoze.what` module which has been merged into TurboGears itself to make easier to support different authentication backends.

A `predicate` is the condition that must be met for the user to be able to access the requested source. Such a predicate, or condition, may be made up of more predicates – those are called *compound predicates*. Action controllers, or controllers, may have only one predicate, be it single or compound.

A `predicate checker` is a class that checks whether a predicate or condition is met.

If a user is not logged in, or does not have the proper permissions, the predicate checker throws a 401 (HTTP Unauthorized) which is caught by the `repoze.who` middleware to display the login page allowing the user to login, and redirecting the user back to the proper page when they are done.

For example, if you have a predicate which is “grant access to any authenticated user”, then you can use the following built-in predicate checker:

```
from tg.predicates import not_anonymous

p = not_anonymous(msg='Only logged in users can read this post')
```

Or if you have a predicate which is “allow access to root or anyone with the ‘manage’ permission”, then you may use the following built-in predicate checker:

```
from tg.predicates import Any, is_user, has_permission

p = Any(is_user('root'), has_permission('manage'),
        msg='Only administrators can remove blog posts')
```

As you may have noticed, predicates receive the `msg` keyword argument to use its value as the error message if the predicate is not met. It's optional and if you don't define it, the built-in predicates will use the default English message; you may take advantage of this functionality to make such messages translatable.

Note: Good predicate messages don't explain *what* went wrong; instead, they describe the predicate in the current context (regardless of whether the condition is met or not!). This is because such messages may be used in places other than in a user-visible message (e.g., in the log file).

- Really bad: “Please login to access this area”.
- Bad: “You cannot delete an user account because you are not an administrator”.
- OK: “You have to be an administrator to delete user accounts”.

- Perfect: “Only administrators can delete user accounts”.

Below are described the convenient utilities TurboGears provides to deal with predicates in your applications.

Action-level authorization

You can control access on a per action basis by using the `tg.decorators.require()` decorator on the actions in question. All you have to do is pass the predicate to that decorator. For example:

```
# ...
from tg import require
from tg.predicates import Any, is_user, has_permission
# ...
class MyCoolController(BaseController):
    # ...
    @expose('yourproject.templates.start_vacations')
    @require(Any(is_user('root'), has_permission('manage'),
                msg='Only administrators can remove blog posts'))
    def only_for_admins():
        flash('Hello admin!')
        dict()
    # ...
```

Controller-level authorization

If you want that all the actions from a given controller meet a common authorization criteria, then you may define the `allow_only` attribute of your controller class:

```
from yourproject.lib.base import BaseController

class Admin(BaseController):
    allow_only = predicates.has_permission('manage')

    @expose('yourproject.templates.index')
    def index(self):
        flash(_("Secure controller here"))
        return dict(page='index')

    @expose('yourproject.templates.index')
    def some_where(self):
        """This is protected too.

        Only those with "manage" permissions may access.

        """
        return dict()
```

If you need to specify additional options for the requirement you can assign a `require` instance instead of a plain predicate to the `allow_only` attribute:

```
from yourproject.lib.base import BaseController
from tg import require

class AdminAPI(BaseController):
    allow_only = require(predicates.has_permission('manage'),
                        smart_denial=True)
```

```
@expose('json')
def create_user(self, user_name):
    pass
```

Warning: Do not use this feature if the login URL would be mapped to that controller, as that would result in a *cyclic redirect*.

Built-in predicate checkers

These are the predicate checkers that are included with `tg.predicates`, although the list below may not always be up-to-date:

Single predicate checkers

class `tg.predicates.not_anonymous`

Check that the current user has been authenticated.

class `tg.predicates.is_user` (*user_name*)

Check that the authenticated user's user name is the specified one.

Parameters `user_name` (*str*) – The required user name.

class `tg.predicates.in_group` (*group_name*)

Check that the user belongs to the specified group.

Parameters `group_name` (*str*) – The name of the group to which the user must belong.

class `tg.predicates.in_all_groups` (*group1_name*, *group2_name* [, *group3_name* ...])

Check that the user belongs to all of the specified groups.

Parameters

- **group1_name** – The name of the first group the user must belong to.
- **group2_name** – The name of the second group the user must belong to.
- ... (*group3_name*) – The name of the other groups the user must belong to.

class `tg.predicates.in_any_group` (*group1_name* [, *group2_name* ...])

Check that the user belongs to at least one of the specified groups.

Parameters

- **group1_name** – The name of the one of the groups the user may belong to.
- ... (*group2_name*) – The name of other groups the user may belong to.

class `tg.predicates.has_permission` (*permission_name*)

Check that the current user has the specified permission.

Parameters `permission_name` – The name of the permission that must be granted to the user.

class `tg.predicates.has_all_permissions` (*permission1_name*, *permission2_name* [, *permission3_name* ...])

Check that the current user has been granted all of the specified permissions.

Parameters

- **permission1_name** – The name of the first permission that must be granted to the user.

- **permission2_name** – The name of the second permission that must be granted to the user.
- ... (*permission3_name*) – The name of the other permissions that must be granted to the user.

class `tg.predicates.has_any_permission` (*permission1_name* [, *permission2_name* ...])
Check that the user has at least one of the specified permissions.

Parameters

- **permission1_name** – The name of one of the permissions that may be granted to the user.
- ... (*permission2_name*) – The name of the other permissions that may be granted to the user.

class `tg.predicates.Not` (*predicate*)
Negate the specified predicate.

Parameters *predicate* – The predicate to be negated.

Custom single predicate checkers

You may create your own predicate checkers if the built-in ones are not enough to achieve a given task.

To do so, you should extend the `tg.predicates.Predicate` class. For example, if your predicate is “Check that the current month is the specified one”, your predicate checker may look like this:

```
from datetime import date
from tg.predicates import Predicate

class is_month(Predicate):
    message = 'The current month must be %(right_month)s'

    def __init__(self, right_month, **kwargs):
        self.right_month = right_month
        super(is_month, self).__init__(**kwargs)

    def evaluate(self, environ, credentials):
        if date.today().month != self.right_month:
            self.unmet()
```

Warning: When you create a predicate, don’t try to guess/assume the context in which the predicate is evaluated when you write the predicate message because such a predicate may be used in a different context.

- Bad: “The software can be released if it’s `%(right_month)s`”.
- Good: “The current month must be `%(right_month)s`”.

If you defined that class in, say, `{yourproject}.lib.auth`, you may use it as in this example:

```
# ...
from spain_travels.lib.auth import is_month
# ...
class SummerVacations(BaseController):
    # ...
    @expose('spain_travels.templates.start_vacations')
    @require(is_month(7))
    def start_vacations():
        flash('Have fun!')
        dict()
    # ...
```

Built-in compound predicate checkers

You may create a *compound predicate* by aggregating single (or even compound) predicate checkers with the functions below:

```
class tg.predicates.All (predicate1, predicate2[, predicate3 ... ])  
    Check that all of the specified predicates are met.
```

Parameters

- **predicate1** – The first predicate that must be met.
- **predicate2** – The second predicate that must be met.
- ... (*predicate3*) – The other predicates that must be met.

```
class tg.predicates.Any (predicate1[, predicate2 ... ])  
    Check that at least one of the specified predicates is met.
```

Parameters

- **predicate1** – One of the predicates that may be met.
- ... (*predicate2*) – Other predicates that may be met.

But you can also nest compound predicates:

```
# ...  
from yourproject.lib.auth import is_month  
# ...  
@authorize.require (authorize.All (  
    Any (is_month(4), is_month(10)),  
    predicates.has_permission('release')  
))  
def release_ubuntu (self, **kwargs):  
    return dict()  
# ...
```

Which translates as “Anyone granted the ‘release’ permission may release a version of Ubuntu, if and only if it’s April or October”.

2.8 Web Session Usage

Status Work in progress

2.8.1 Why Use Sessions?

Sessions are a common way to keep simple browsing data attached to a user’s browser. This is generally used to store simple data that does not need to be persisted in a database.

Sessions in TurboGears can be backed by the filesystem, memcache, the database, or by hashed cookie values. By default, cookies are used for storing the session data, which is only good for storing very little amounts of data in the session since all data will be sent back and forth within the cookie. If you are storing lots of data in the session, *Memcache* is recommended.

Warning: Using cookies for storing the whole session’s content exposes your application to possible exploits if the attacker gets to know the secret key which is used for the encryption of the cookies. Considering this, it is probably better to use the filesystem storage if you don’t want to set up memcache.

Note: When using the filesystem backed storage, you must be aware of the fact, that beaker does **not** clean up the session files at all. You have to make sure to clean up the data directory on a regular basis yourself. Refer to the [Beaker documentation](#) for more details.

2.8.2 How To Use Sessions?

If you just quickstarted a TurboGears 2 application, the session system is pre-configured and ready to be used.

By default we are using the Beaker session system. This system is configured to use hashed cookies for session storage.

Each time a client connects, the session middleware (Beaker) will inspect the cookie using the cookie name we have defined in the configuration file.

If the cookie is not found it will be set in the browser. On all subsequent visits, the middleware will find the cookie and make use of it.

When using the cookie based backend, all data that you put into the session will be pickled, hashed and encrypted by the middleware when sending the response to the browser and vice-versa when reading the request.

In the other backends, the cookie only contains a large random key that was set at the first visit and has been associated behind the scenes to a file in the file system cache. This key is then used to lookup and retrieve the session data from the proper datastore.

OK, enough with theory! Let's get to some real life (sort of) examples. Open up your root controller and add the following import at the top the file:

```
from tg import session
```

What you get is a Session instance that is always request-local, in other words, it's the session for this particular user. The session can be manipulated in much the same way as a standard python dictionary.

Here is how you search for a key in the session:

```
if session.get('mysuperkey', None):  
    # do something intelligent  
    pass
```

and here is how to set a key in the session:

```
session['mysuperkey'] = 'some python data I need to store'  
session.save()
```

You should note that you need to explicitly save the session in order for your keys to be stored in the session.

You can delete all user session with the `delete()` method of the session object:

```
session.delete()
```

Even though it's not customary to delete all user sessions on a production environment, you will typically do it for cleaning up after usability or functional tests.

2.8.3 Avoid automatic session extension

TurboGears by default automatically extends session life time at every request if a session is already available. You can avoid this behavior by changing your application configuration

```
beaker.session.tg_avoid_touch = true
```

This will also prevent TurboGears from causing an automatic session save at every request.

2.9 Caching

Caching is a common technique to achieve performance goals, when a web application has to perform some operation that could take a long time. There are two major types of caching used in Web Applications:

- *Whole-page caching* – works at the HTTP protocol level to avoid entire requests to the server by having either the user’s browser, or an intermediate proxy server (such as Squid) intercept the request and return a cached copy of the file.
- Application-level caching – works within the application server to cache computed values, often the results of complex database queries, so that future requests can avoid needing to re-calculate the values.

Most web applications can only make very selective use of HTTP-level caching, such as for caching generated RSS feeds, but that use of HTTP-level caching can dramatically reduce load on your server, particularly when using an external proxy such as Squid and encountering a high-traffic event (such as the *Slashdot Effect*).

For web applications, application-level caching provides a flexible way to cache the results of complex queries so that the total load of a given controller method can be reduced to a few user-specific or case-specific queries and the rendering overhead of a template. Even within templates, application-level caching can be used to cache rendered HTML for those fragments of the interface which are comparatively static, such as database-configured menus, reducing potentially recursive database queries to simple memory-based cache lookups.

2.9.1 Application-level Caching

TurboGears comes with application-level caching middleware enabled by default in QuickStarted projects. The middleware, *Beaker* is the same package which provides Session storage for QuickStarted projects. Beaker is the cache framework used by TurboGears 2.3.5.

Beaker supports a variety of backends which can be used for cache or session storage:

- memory – per-process storage, extremely fast
- filesystem – per-server storage, very fast, multi-process
- “DBM” database – per-server storage, fairly fast, multi-process
- SQLAlchemy database – per-database-server storage, integrated into your main DB infrastructure, so potentially shared, replicated, etc., but generally slower than memory, filesystem or DBM approaches
- *Memcached* – (potentially) multi-server memory-based cache, extremely fast, but with some system setup requirements

Each of these backends can be configured from your application’s configuration file, and the resulting caches can be used with the same API within your application.

Controller Caching

TurboGears provides an helper for quick controller caching. By using the `tg.decorators.cached()` decorator on a controller you can specify that the whole controller body has to be cached depending on the parameters of the request.

By default also the `Content-Type` and `Content-Length` headers will be cached, so that we ensure that the content related headers get restored together with the cached content.

Using `@cached` is pretty straightforward:

```
class SimpleController(TGController):

    @cached(expire=100, type='memory')
    @expose()
    def simple(self):
        return "Hey, this is a cached controller!"
```

You can specify additional headers to be cached or change the cache key by specifying the `@cached` arguments. Please refer to `tg.decorators.cached()` reference for details.

Note: Please note that `@cached` will only cache the controller body, the template will still be rendered. To cache both the controller and template join it with `template_cache`

Manually Using the Cache

The configured *Beaker* is more properly thought of as a *CacheManager*, as it provides access to multiple independent cache namespaces.

To access the cache from within a controller module:

```
from tg import cache

@expose()
def some_action(self, day):
    # hypothetical action that uses a 'day' variable as its key

    def expensive_function():
        # do something that takes a lot of cpu/resources
        return expensive_call()

    # Get a cache for a specific namespace, you can name it whatever
    # you want, in this case its 'my_function'
    mycache = cache.get_cache('my_function')

    # Get the value, this will create the cache copy the first time
    # and any time it expires (in seconds, so 3600 = one hour)
    cachedvalue = mycache.get_value(
        key=day,
        createfunc=expensive_function,
        expiretime=3600
    )
    return dict(myvalue=cachedvalue)
```

The *Beaker* cache is a two-level namespace, with the keys at each level being string values. The call to `cache.get_cache()` retrieves a cache namespace which will map a set of string keys to stored values. Each value that is stored in the cache must be *pickle-able*.

Pay attention to the keys you are using to store your cached values. You need to be sure that your keys encode all of the information that the results being cached depend upon in a unique manner. In the example above, we use *day* as the key for our cached value, on the assumption that this is the only value which affects the calculation of *expensive_function*, if there were multiple parameters involved, we would need to encode each of them into the key.

Note: The *Beaker* API exposed here requires that your functions for calculating complex values be callables taking 0 arguments. Often you will use a nested function to provide this interface as simply as possible. This function will only

be called if there is a *cache miss*, that is, if the cache does not currently have the given key recorded (or the recorded key has expired).

Other Cache Operations

The cache also supports the removal values from the cache, using the key(s) to identify the value(s) to be removed and it also supports clearing the cache completely, should it need to be reset.

```
# Clear the cache
mycache.clear()

# Remove a specific key
mycache.remove_value('some_key')
```

Configuring Beaker

Beaker is configured in your QuickStarted application's main configuration file in the `app:main` section.

To use memory-based caching:

```
[app:main]
beaker.cache.type = memory
```

To use file-based caching:

```
[app:main]
beaker.cache.type = file
beaker.cache.data_dir = /tmp/cache/beaker
beaker.cache.lock_dir = /tmp/lock/beaker
```

To use DBM-file-based caching:

```
[app:main]
beaker.cache.type = dbm
beaker.cache.data_dir = /tmp/cache/beaker
beaker.cache.lock_dir = /tmp/lock/beaker
```

To use SQLAlchemy-based caching you must provide the `url` parameter for the *Beaker* configuration. This can be any valid SQLAlchemy URL, the *Beaker* storage table will be created by *Beaker* if necessary:

```
[app:main]
beaker.cache.type = ext:database
beaker.cache.url = sqlite:///tmp/cache/beaker.sqlite
```

Memcached

Memcached allows for creating a pool of collaborating servers which manage a single distributed cache which can be shared by large numbers of front-end servers (i.e. TurboGears instances). Memcached can be extremely fast and scales up very well, but it involves an external daemon process which (normally) must be maintained (and secured) by your sysadmin.

Memcached is a system-level daemon which is intended for use solely on “trusted” networks, there is little or no security provided by the daemon (it trusts anyone who can connect to it), so you should never run the daemon on a network which can be accessed by the public! To repeat, do *not* run memcached without a firewall or other network partitioning

mechanism! Further, be careful about storing any sensitive or authentication/authorization data in memcache, as any attacker who can gain access to the network can access this information.

Ubuntu/Debian servers will generally have memcached configured by default to only run on the localhost interface, and will have a small amount of memory (say 64MB) configured. The `/etc/memcached.conf` file can be edited to change those parameters. The memcached daemon will also normally be deactivated by default on installation. A basic memcached installation might look like this on an Ubuntu host:

```
sudo apt-get install memcached
sudo vim /etc/default/memcached
# ENABLE_MEMCACHED=yes
sudo vim /etc/memcached.conf
# Set your desired parameters...
sudo /etc/init.d/memcached restart
# now install the Python-side client library...
# note that there are other implementations as well...
easy_install python-memcached
```

You then need to configure TurboGears/Pylon's beaker support to use the memcached daemon in your `.ini` files:

```
[app:main]
beaker.cache.type = ext:memcached
beaker.cache.url = 127.0.0.1:11211
# you can also store sessions in memcached, should you wish
# beaker.session.type = ext:memcached
# beaker.session.url = 127.0.0.1:11211
```

You can have multiple memcached servers specified using `;` separators. Usage, as you might imagine is the same as with any other *Beaker* cache configuration (that is, to some extent, the point of the Beaker Cache abstraction, after all):

References

- [Beaker Caching](#) – discussion of use of Beaker's caching services
- [Beaker Configuration](#) – the various parameters which can be used to configure Beaker in your config files
- [Memcached](#) – the memcached project
- [Python Memcached](#) – Python client-side binding for memcached
- [Caching for Performance](#) – Stephen Pierzchala's general introduction to the concept of caching in order to improve web-site performance

2.9.2 Template Caching

Genshi Loader Cache

genshi will retrieve the templates from a cache if they have not changed. This cache has a default size of 25, when there are more than 25, the least recently used templates will be removed from this cache.

You can change this behavior by setting the `genshi.max_cache_size` option into the `development.ini`:

```
[app:main]
genshi.max_cache_size=100
```

Another speed boost can be achieved by disabling template automatic reloading.

```
[app:main]
auto_reload_templates = false
```

Prerendered Templates Caches

In templates, the cache namespace will automatically be set to the name of the template being rendered. To cache a template you just have to return the `tg_cache` option from the controller that renders the cached template.

`tg_cache` is a dictionary that accepts the following keys:

- `key`: The cache key. Default: None
- `expire`: how long the cache must stay alive. Default: never expires
- `type`: memory, dbm, memcached. Default: dbm

if any of the keys is available the others will default, if all three are missing caching will be disabled. For example to enable caching for 1 hour for the profile of an user:

```
@expose('myproj.templates.profile')
def profile(self, username):
    user = DBSession.query(User).filter_by(user_name=username).first()
    return dict(user=user, tg_cache=dict(key=username, expire=3600))
```

2.9.3 HTTP-Level Caching

HTTP supports caching of whole responses (web-pages, images, script-files and the like). This kind of caching can dramatically speed up web-sites where the bulk of the content being served is largely static, or changes predictably, or where some commonly viewed page (such as a home-page) requires complex operations to generate.

HTTP-level caching is handled by external services, such as a [Squid](#) proxy or the user's browser cache. The web application's role in HTTP-level caching is simply to signal to the external service what level of caching is appropriate for a given piece of content.

Note: If *any* part of you page has to be dynamically generated, even the simplest fragment, such as a user-name, for each request HTTP caching likely will not work for you. Once the page is HTTP-cached, the application server will not receive any further requests until the cache expires, so it will not generally be able to do even minor customizations.

Browser-side Caching with ETag

HTTP/1.1 supports the *ETag* caching system that allows the browser to use its own cache instead of requiring regeneration of the entire page. ETag-based caching avoids repeated generation of content but if the browser has never seen the page before, the page will still be generated. Therefore using ETag caching in conjunction with one of the other types of caching listed here will achieve optimal throughput and avoid unnecessary calls on resource-intensive operations.

Caching via ETag involves sending the browser an ETag header so that it knows to save and possibly use a cached copy of the page from its own cache, instead of requesting the application to send a fresh copy.

The `etag_cache()` function will set the proper HTTP headers if the browser doesn't yet have a copy of the page. Otherwise, a 304 HTTP Exception will be thrown that is then caught by Paste middleware and turned into a proper 304 response to the browser. This will cause the browser to use its own locally-cached copy.

ETag-based caching requires a single key which is sent in the ETag HTTP header back to the browser. The [RFC specification for HTTP headers](#) indicates that an ETag header merely needs to be a string. This value of this string

does not need to be unique for every URL as the browser itself determines whether to use its own copy, this decision is based on the URL and the ETag key.

```
from tg.controllers.util import etag_cache
def my_action(self):
    etag_cache('somekey')
    return render('/show.myt', cache_expire=3600)
```

Or to change other aspects of the response:

```
from tg.controllers.util import etag_cache
from tg import response
def my_action(self):
    etag_cache('somekey')
    response.headers['content-type'] = 'text/plain'
    return render('/show.myt', cache_expire=3600)
```

Note: In this example that we are using template caching in addition to ETag caching. If a new visitor comes to the site, we avoid re-rendering the template if a cached copy exists and repeat hits to the page by that user will then trigger the ETag cache. This example also will never change the ETag key, so the browsers cache will always be used if it has one.

The frequency with which an ETag cache key is changed will depend on the web application and the developer's assessment of how often the browser should be prompted to fetch a fresh copy of the page.

ETag From Wikipedia An ETag (entity tag) is an HTTP response header returned by an HTTP/1.1 compliant web server used to determine change in content at a given URL.

Todo

Add links to Beaker region (task-specific caching mechanisms) support.

Todo

Document what the default Beaker cache setup is for TG 2.3.5 quickstarted projects (file-based, likely).

Todo

Provide code-sample for use of cache within templates

2.10 Handling Internationalization And Localization

Status Work in progress

Table of Contents

- [Handling Internationalization And Localization](#)
 - [Language Auto-Select](#)
 - [Forcing a Language](#)
 - [Making your code international](#)
 - [An i18n Quick Start](#)
 - [Commands](#)

Turbogears2 relies on Babel for i18n and l10n support. So if this document is not enough you will want to check their respective documentation:

- Babel's [UserGuide](#)

A quickstarted project comes fully i18n enabled so you should get started quickly.

If you're lucky enough you'll even see "Your application is now running" message in your language.

2.10.1 Language Auto-Select

Turbogears2 contains the logic to setup request's language based on browser's preferences(*).

[*] - Every modern browser sends a special header along with every web request which tells the server which language it would prefer to see in a response.

The language in use during the request is available through the `tg.i18n.get_lang()` function. This will report the currently selected languages both in case of an auto-detected language preference or in case of languages forced by the developer.

Languages returned by `tg.i18n.get_lang()` are ordered by user preference. By default all the requested languages are returned, if you want to get only the ones your application supports call `get_lang(all=False)`.

The current language in use is usually `get_lang(all=False)[0]`.

2.10.2 Forcing a Language

Developer can force the currently used language for each request using the `tg.i18n.set_temporary_lang()` function. This will change the language only for the current request.

If you need to permanently change the language for the user session duration, `tg.i18n.set_lang()` can be used. If TurboGears session support is enabled it will store the choosen language inside the session and recover it whenever the user comes back on next request.

2.10.3 Making your code international

Whenever you write a message that has to displayed you must let TurboGears know that it has to be translated.

Even though TurboGears is able to automatically detect content inside tags and mark them for translation all the strings inside controllers must be explicitly marked as translated.

This can be achieved with the `tg.i18n.gettext()` and `tg.i18n.lazy_gettext()` calls which are usually imported with `_` and `l_` names:

```
from tg.i18n import gettext as _

class RootController(BaseController):
    @expose('myproj.templates.index')
    def index(self):
        return dict(msg=_('Hello World'))
```

In the previous example the 'Hello World' string will be detect by TurboGears when collecting translatable text and will display in the browser language if a translation for that language is available.

While `gettext()` works perfectly to translate strings inside a request it does not allow translating strings outside a request. This is due to the fact that TurboGears won't know the browser language when there isn't a running request. To translate global variables, parameters default values or any other string which is created outside a controller method the `lazy_gettext` method must be used:

```

from tg.i18n import lazy_ugettext as l_

class RootController(BaseController):
    @expose('myproj.templates.index')
    def index(self, msg=l_('Hello World')):
        return dict(msg=msg)

```

In this case the `msg` parameter is translated using `lazy_ugettext()` as it is constructed at controller import time when no request is available. This will create an object that will translate the given string only when the string itself is displayed or evaluated.

Keep in mind that as the lazy string object built by `lazy_ugettext()` is translated whenever the string is evaluated joining strings or editing it will force the translation. So the resulting object must still be evaluated only inside a request or it will always be translated to the default project language only.

2.10.4 An i18n Quick Start

After quickstarting your project, you will notice that the `setup.py` file contains the following section:

```

message_extractors = {'yourapp': [
    ('**.py', 'python', None),
    ('templates/**/*.mako', 'mako', None),
    ('templates/**/*.html', 'genshi', None),
    ('public/**', 'ignore', None)]},

```

This is an extraction method mapping that indicates to distutils which files should be searched for strings to be translated. TurboGears2 uses Babel to extract messages to a `.pot` file in your project's `i18n` directory. Don't forget to add it to your revision control system if you use one.

1. Extract all the translatable strings from your project's files by using the following command:

```
python setup.py extract_messages
```

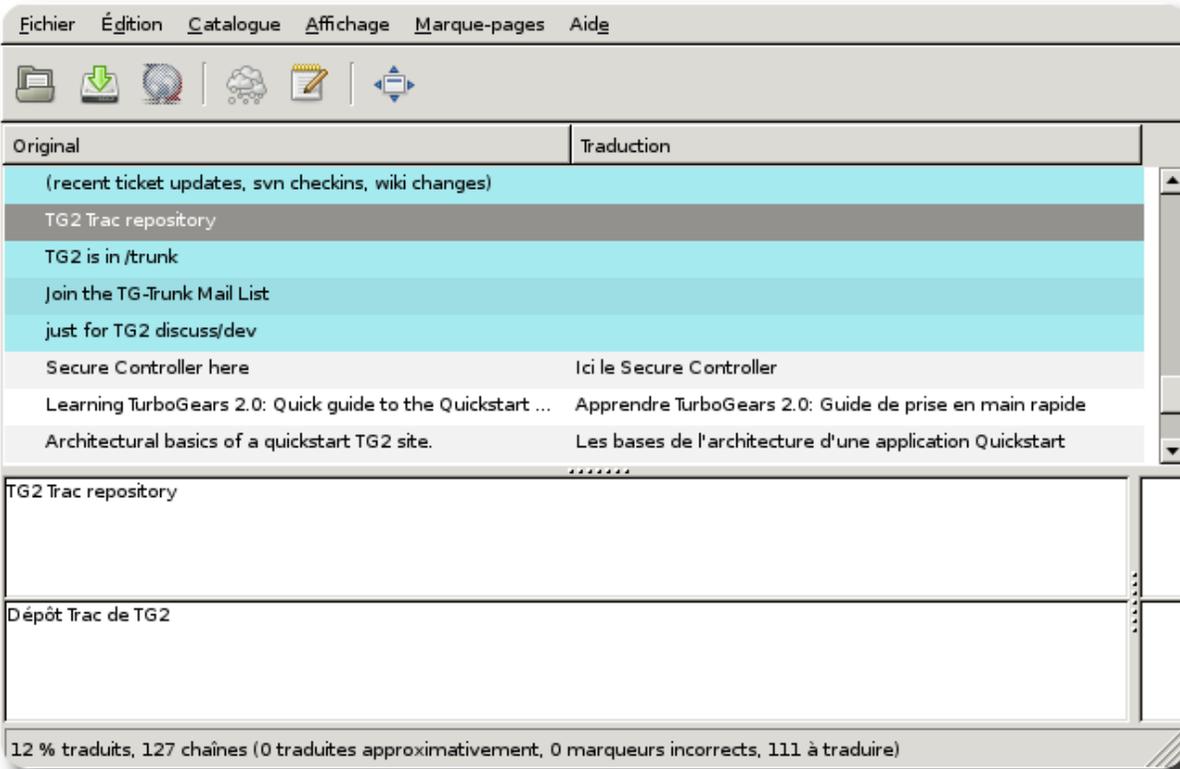
This command will generate a "pot" file in the `i18n` folder of your application. This pot file is the reference file that serves for all the different translations.

2. Create a translation catalog for your language, let's take 'zh_tw' for example:

```
python setup.py init_catalog -l zh_tw
```

3. Edit your language in `i18n/[country code]/LC_MESSAGES/[project-name].po`

If you're not an expert in i18n or if you would like to give the files to someone else so that he helps you we recommend that you use the really nice poedit program. This program works nicely on GNU/Linux and Windows and provides a nice user-interface to edit po files.



4. Compile your lang:

```
python setup.py compile_catalog
```

5. Config development.ini:

```
[app:main]
use = egg: my-project
full_stack = true
lang = zh_tw
```

6. Start server:

```
gearbox serve --reload
```

And see the local message show on the screen.

2.10.5 Commands

To fresh start a translation, you could use the following command to handle your locales:

init_catalog

You can extract all messages from the project with the following command:

```
python setup.py init_catalog -l [country code]
```

The country code could be es(Spanish), fr(France), zh_tw(Taiwan), jp(JAPAN), ru(Russian), or any other country code.

Compile Catalog

You can extract all messages from the project with the following command:

```
python setup.py compile_catalog
```

Update Catalog

You can update the catalog with the following command:

```
python setup.py update_catalog
```

2.11 Writing TurboGears Extensions

TurboGears provides a bunch of hook points and ways to extend the framework behavior to write extensions. Most of the needs can usually be satisfied by relying on `tg.configuration.AppConfig`, *Configuration Milestones* and *Hooks*.

2.11.1 Creating an Extension

TurboGears extensions are identified by the `tgext.*` package name, since version 2.3.2 the devtools provide a command to quickly create a new TurboGears extension which can be both used by itself or plugged through `tgext.pluggable`.

To quickstart the extension run:

```
$ gearbox tgext -n myextension -a "My Name" -e "my@email.com"
```

This will create a `tgext.myextension` directory which a simple sample extension inside. To tune the author, description, license of your extension have a look at the `tgext` command options using `gearbox help tgext`.

2.11.2 Tuning on the Extension

If you created the `tgext.myextension` using the `tgext` command you can quickly turn it on by editing your application `config/app_cfg.py` and appending at the end the following lines:

```
import tgext.myextension
tgext.myextension.plugme(base_config)
```

of through the *pluggable application* interface if `tgext.pluggable` is available:

```
from tgext.pluggable import plug
plug(base_config, 'tgext.myextension')
```

By enabling the autogenerated extension and starting `gearbox serve` again, you will notice that it just provides a bunch of logging hooks and not much more:

```
17:52:30,019 INFO [tgext.myextension] Setting up tgext.myextension extension...
17:52:30,023 INFO [tgext.myextension] >>> Public files path is /home/USER/myapp/myapp/public
17:52:30,023 INFO [tgext.myextension] + Application Running!
17:52:30,040 INFO [gearbox] Starting server in PID 22484.
Starting HTTP server on http://0.0.0.0:8080
17:52:30,740 INFO [tgext.myextension] Serving: /
```

2.11.3 The Sample Extension

The sample extension created by the `tgext` command provides its actual setup inside the `__init__.py` file. This file showcases some of the extension points provided by TurboGears.

If you actually created `tgext.myextension` you should get a `tgext.myextension/tgext/myextension/__init__.py` file with the following content:

```
from tg import config
from tg import hooks
from tg.configuration import milestones

import logging
log = logging.getLogger('tgext.myextension')

# This is the entry point of your extension, will be called
# both when the user plugs the extension manually or through tgext.pluggable
# What you write here has the same effect as writing it into app_cfg.py
# So it is possible to plug other extensions you depend on.
def plugme(configurator, options=None):
    if options is None:
        options = {}

    log.info('Setting up tgext.myextension extension...')
    milestones.config_ready.register(SetupExtension(configurator))

    return dict(appid='tgext.myextension')

# Most of your extension initialization should probably happen here,
# where it's granted that .ini configuration file has already been loaded
# in tg.config but you can still register hooks or other milestones.
class SetupExtension(object):
    def __init__(self, configurator):
        self.configurator = configurator

    def __call__(self):
        log.info('>>> Plugin files path is %s' % config['paths']['static_files'])
        hooks.register('startup', self.on_startup)

    def echo_wrapper_factory(handler, config):
        def echo_wrapper(controller, environ, context):
            log.info('Serving: %s' % context.request.path)
            return handler(controller, environ, context)
        return echo_wrapper

    # Application Wrappers are much like easier WSGI Middleware
    # that get a TurboGears context and return a Response object.
    self.configurator.register_wrapper(echo_wrapper_factory)

    def on_startup(self):
        log.info('+ Application Running!')
```

The core parts of the previous example are:

- **plugme function, this is the function used to turn on your extension.** will be automatically called by `tgext.pluggable` when the extension is enabled using the *pluggable application* interface or by the user itself when manually enabling your extension. Inside this method the application configurator

(AppConfig) object is available and the options the user specified for the extension, but not application configuration as it has not been loaded yet.

- **SetupExtension.__call__**, this is a callable that is registered by the `plugme` function for the `config_read` milestone so that it is executed when the `.ini` configuration has been loaded and merged with the options declared through the application configurator in `config/app_cfg.py`.

Here you can register additional milestones, functions or access and modify the application configurator through the `self.configurator` object.

- **SetupExtension.on_startup** This is a sample hook registered on application startup by `SetupExtension.__call__` that says hello when the application has started. Have a look at [Hooks](#) for a complete list of available hooks.

2.11.4 Extensions with models and controllers

If your extension needs to expose models and controllers you probably want to have a look at **Pluggable Applications** which are meant to create reusable turbogears applications that can be plugged inside other applications to extend their features.

2.12 Pluggable Applications with TurboGears

TurboGears 2.1.4 introduced support for pluggable applications using `tgext.pluggable`. `tgext.pluggable` is now the official supported way in TurboGears to create pluggable reusable applications. Currently only SQLAlchemy based applications are supported as pluggable applications.

Official documentation for `tgext.pluggable` can be found at: <http://pypi.python.org/pypi/tgext.pluggable>

2.12.1 Supported Features

Pluggable applications can define their own:

- **controllers**, which will be automatically mounted when the application is purged.
- **models**, which will be available inside and outside of the plugged application.
- **helpers**, which can be automatically exposed in `h` object in application template.
- **bootstrap**, which will be executed when `setup-app` is called.
- **statics**, which will be available at their own private path.

2.12.2 Mounting a pluggable application

In your application `config/app_cfg.py` import `plug` from `tgext.pluggable` and call it for each pluggable application you want to enable.

The plugged package must be installed in your environment.

```
from tgext.pluggable import plug
plug(base_config, 'package_name')
```

2.12.3 Creating Pluggable Applications

`tgext.pluggable` provides a **quickstart-pluggable** command to create a new pluggable applications:

```
$ gearbox quickstart-pluggable plugtest
...
```

The quickstarted application will provide an example on how to use models, helpers, bootstrap, controllers and statics.

2.13 Pagination in TurboGears

2.13.1 Paginate Decorator

TurboGears provides a convenient `paginate()` decorator that you can combine with `expose()`. To use it, you simply have to pass it the name of a collection to paginate. In `controller/root.py`:

```
from tg.decorators import paginate

@expose("paginatesample.templates.movie_list_deco")
@paginate("movies", items_per_page=5)
def decolist(self):
    """
    List and paginate all movies in the database using the
    paginate() decorator.
    """

    movies = DBSession.query(Movie)
    return dict(movies=movies, page='paginatesample Movie list')
```

In your template, you can now use the collection direction since it will be trimmed to only contain the current page. You will also have a basic page navigation with `${tmpl_context.paginators.movies.pager() }`:

```
<ol>
  <li py:for="movie in movies" py:content="movie">Movie title and year</li>
</ol>

<p class="pagelist">
  ${tmpl_context.paginators.movies.pager() }
</p>
```

2.13.2 Advanced Pagination

The `pager` method of the paginator supports various customization options to tune the look and feel of the paginator, make sure you take a look at `tg.support.paginate.Page` for more details:

```
 ${tmpl_context.paginators.movies.pager(format='~3~', page_param='page', show_if_single_page=True)}
```

Adding Parameters to Links

You can pass any number of arguments to the `pager` function and they will be used to create the links to the other pages.

For example with the following code:

```
${tmpl_context.paginators.movies.pager(param1='hi', param2='man')}
```

the resulting links will be:

- `/list?page=1¶m1=hi¶m2=man`
- `/list?page=2¶m1=hi¶m2=man`

and so on...

By default the url used to generate links will be the same of the page where the paginated data will be visible, this can be changed by passing the **link** argument to the `pager` function:

```
${tmpl_context.paginators.movies.pager(link='/otherlink', param1='hi', param2='man')}
```

and the resulting link will be generated by using the provided url:

- `/otherlink?page=1¶m1=hi¶m2=man`

Adding Previous And Next Links

Apart from providing the `pager` method the paginator Page object we receive inside our template context provides the `previous_page` and `next_page` properties which can be used to create previous/next links:

```
<p class="pagelist">
  <a class="prevPage" href="/list?page=${tmpl_context.paginators.movies.previous_page}">&lt;&lt;
  ${tmpl_context.paginators.movies.pager(format='~3~', page_param='page', show_if_single_page=
  <a class="nextPage" href="/list?page=${tmpl_context.paginators.movies.next_page}">&gt;&gt;&gt;
</p>
```

Adding Some Arrow Images

Once you added your own previous/next page entities you can style them as you prefer, one common need is to display an image instead of the text:

```
a.prevPage {
  background: url("/images/icons/png/32x32/arrow-left.png") no-repeat;
  padding-left: 18px;
  padding-right: 18px;
  padding-top: 12px;
  padding-bottom: 15px;
  text-decoration: none;
}

a.nextPage {
  background: url("/images/icons/png/32x32/arrow-right.png") no-repeat;
  padding-left: 18px;
  padding-right: 18px;
  padding-top: 12px;
  padding-bottom: 15px;
  text-decoration: none;
}
```

2.14 Using MongoDB

TurboGears supports [MongoDB](#) out of the box by using the [Ming](#) ORM. [Ming](#) was made to look like SQLAlchemy, so if you are proficient with SQLAlchemy and MongoDB it should be easy for you to get used to the [Ming](#) query language. This also makes easy to port a TurboGears SQLAlchemy based application to MongoDB.

2.14.1 QuickStarting with MongoDB

To create a project using [MongoDB](#) you just need to pass the `--ming` option to the `gearbox quickstart` command.

```
$ gearbox quickstart --ming
```

The quickstarted project will provide an authentication and authorization layer like the one that is provided for the SQLAlchemy version. This means that you will have the same users and groups you had on the standard quickstarted project and also that all the predicates to check for authorization should work like before.

The main difference is that you won't be able to use the application without having a running [MongoDB](#) database on the local machine.

By default the application will try to connect to a server on port `27017` on local machine using a database that has the same name of your package.

This can be changed by editing the `development.ini` file:

```
ming.url = mongodb://localhost:27017/  
ming.db = myproject
```

Now that everything is in place to start using [MongoDB](#) as your database server you just need to proceed the usual way by filling your database.

```
$ gearbox setup-app
```

The quickstart command from above will create the authentication collections and setup a default user/password for you:

```
user: manager  
password: managepass
```

2.14.2 Working With Ming

If you don't know how [Ming](#) works at all, please take a few minutes to read over these tutorials:

- [ORM Tutorial](#) – which covers the ORM parts
- [Intro to Ming](#) – which covers a more general intro

Your quickstarted project will have a subpackage called *model*, made up of the following files:

- `__init__.py`: This is where the database access is set up. Your collections should be imported into this module, and you're highly encouraged to define them in a separate module - *entities*, for example.
- `session.py`: This file defines the session of your database connection. By default TurboGears will use a Session object with multithreading support. You will usually need to import this each time you have to declare a `MappedClass` to specify the session that has to be used to perform queries.

- *auth.py*: This file will be created if you enabled authentication and authorization in the quickstart. It defines two collections `repoze.what.quickstart` relies on: *User* (for the registered members in your website and the groups they belong to) and *Permission* (a permission granted to one or more groups).

2.14.3 Defining Your Own Collections

By default TurboGears configures [Ming](#) in Declarative mode. This is similar to the SQLAlchemy declarative support and needs each model to inherit from the `MappedClass` class.

The tables defined by the quickstart in *model/auth.py* are based on the declarative method, so you may want to check it out to see how columns are defined for these tables. For more information, you may read the [ORM Tutorial](#).

Once you have defined your collections in a separate module in the *model* package, they should be imported from *model/__init__.py*. So the end of this file would look like this:

```
# Import your model modules here.
from auth import User, Permission
# Say you defined these three classes in the 'movies'
# module of your 'model' package.
from movies import Movie, Actor, Director
```

Indexing Support

TurboGears supports also automatic indexing of [MongoDB](#) fields. If you want to guarantee that a field is unique or indexed you just have to specify the `unique_indexes` or `indexes` variables for the `__mongometa__` attribute of the mapped class.

```
class Permission(MappedClass):
    class __mongometa__:
        session = DBSession
        name = 'tg_permission'
        unique_indexes = [('permission_name',),]
```

TurboGears will ensure indexes for your each time the application is started, this is performed inside the `init_model` function.

2.14.4 Handling Relationships

Ming comes with support to one-to-many and many-to-one [Relations](#) they provide an easy to use access to related objects. The fact that this relation is read only isn't a real issue as the related objects will have a `ForeignIdProperty` which can be changed to add or remove objects to the relation.

As MongoDB provides too many ways to express a many-to-many relationship, those kind of relations are instead left on their own. TurboGears anyway provides a tool to make easier to access and modify those relationships.

`tgming.ProgrammaticRelationProperty` provides easy access to those relationships exposing them as a list while leaving to the developer the flexibility to implement the relationship as it best suites the model.

A good example of how the `ProgrammaticRelationProperty` works is the `User` to `Group` relationship:

```
from tgming import ProgrammaticRelationProperty

class Group(MappedClass):
    class __mongometa__:
        session = DBSession
        name = 'tg_group'
```

```
group_name = FieldProperty(s.String)

class User(MappedClass):
    class __mongometa__:
        session = DBSession
        name = 'tg_user'

    _groups = FieldProperty(s.Array(str))

    def _get_groups(self):
        return Group.query.find(dict(group_name={'$in':self._groups})).all()
    def _set_groups(self, groups):
        self._groups = [group.group_name for group in groups]
    groups = ProgrammaticRelationProperty(Group, _get_groups, _set_groups)
```

In this case each user will have one or more groups stored with their `group_name` inside the `User._groups` array. Accessing `User.groups` will provide a list of the groups the user is part of. This list is retrieved using `User._get_groups` and can be set with `User._set_groups`.

2.14.5 Using Synonyms

There are cases when you will want to adapt a value from the database before loading and storing it. A simple example of this case is the password field, this will probably be encrypted with some kind of algorithm which has to be applied before saving the field itself.

To handle those cases TurboGears provides the `tgming.SynonymProperty` accessor. This provides a way to hook two functions which have to be called before storing and retrieving the value to adapt it.

```
from tgming import SynonymProperty

class User(MappedClass):
    class __mongometa__:
        session = DBSession
        name = 'tg_user'

    _password = FieldProperty(s.String)

    def _set_password(self, password):
        self._password = self._hash_password(password)
    def _get_password(self):
        return self._password
    password = SynonymProperty(_get_password, _set_password)
```

In the previous example the password property is stored encrypted inside the `User._password` field but it is accessed using the `User.password` property which encrypts it automatically before setting it.

Advanced Documentation

3.1 Installing TurboGears2

This chapter provides more complete information about the install process. In case you had issues installing TurboGears, this is the place to take a look at.

3.1.1 virtualenv and You: A Perfect Match

`virtualenv` is an extremely handy tool while doing development of any sort, or even just testing out a new application. Using it allows you to have a sandbox in which to work, separate from your system's Python. This way, you can try out experimental code without worrying about breaking another application on your system. It also provides easy ways for you to work on developing the next version of your application without worrying about a conflicting version already installed on your system.

It's a tool that you would do well to learn and use. You could even check out the `virtualenvwrapper` tools by Doug Hellman (though this is not required, and we will not assume you have them installed throughout this book, they are still quite nice to have and use).

Installation on Windows

Open a command prompt, and run:

```
C:\> easy_install virtualenv
```

This will install a binary distribution for you, precompiled for Windows.

Installation on UNIX/Linux/Mac OSX

For these platforms, there exists a large amount of variation in the exact process to install `virtualenv`.

1. Attempt to install via your platform's package manager (for example: `apt-get install python-virtualenv` or `yum install virtualenv`).
2. From the command line, attempt a plain `easy_install` via `easy_install virtualenv`
3. Your platform may need to have the Python header files installed. You will need to work with whatever tools come with your platform to make this happen (for instance, OSX requires the XCode tools to install `virtualenv`, or on Ubuntu, you can use `apt-get install python-dev`). After doing this, `easy_install virtualenv` should work.

If none of these methods work, please feel free to ask on the [mailing list](#) for help, and we'll work through it with you.

virtualenv Notes

When you use virtualenv, you have many options available to you (use `virtualenv --help` from the command line to see the full list). We are only going to cover basic use here.

The first thing you need to know is that virtualenv is going to make a directory which amounts to a private installation of Python. This means it will have `bin`, `include`, and `lib` directories. Most commonly, you will be using the files in the `bin` directory: specifically, the new command `activate` will become your best friend.

The second thing you need to know is that you will *rarely* want to use the system `site-packages` directory, and we **never** recommend it with TurboGears2. As a result, we always recommend turning it off when using virtualenv. It makes debugging much easier when you know what is there all the time.

The last thing to note is that you have the option of choosing a different default Python interpreter for your virtualenv. This will allow you to test on different Python versions, such as 2.4, 2.5, 2.6, 2.7, or even [PyPy](#). Normally, this won't matter, but it is helpful to know that you can switch easily.

Usage of virtualenv

To use virtualenv, you run it like so:

```
$ virtualenv --no-site-packages -p /usr/bin/python2.7 ${HOME}/tg2env
```

When done, with these options, you will now have a Python 2.7 virtualenv located at `${HOME}/tg2` that has nothing but what comes with Python. By changing `/usr/bin/python2.7` to point to a different Python interpreter, you will be able to choose a different version of Python. By changing `${HOME}/tg2env` to point to a different directory, you can choose a different location for your new virtualenv.

For the duration of this book, we will assume that the virtualenv you are using is located at `${HOME}/tg2env`. Please change the commands we give to you to match your system's directory structure if you choose to use a different directory for your virtualenv.

Once you have a virtualenv, you must activate it. On a UNIX/Linux/Mac OSX machine, from the command line, you do the following:

```
$ source ${HOME}/tg2env/bin/activate
```

On Windows systems, from the command line, you do the following:

```
C:\> \path\to\virtualenv\Script\activate.bat
```

That's it. From this point onward, any `pip` commands will automatically use your virtualenv, as will your `setup.py` scripts that will be developed in later chapters.

When you are done with this virtualenv, use the command `deactivate`. This will return your environment to what it was, and allow you to work with the system wide Python installation.

3.1.2 Installing TurboGears2

TurboGears2 is actually distributed in two separate packages: TurboGears2 and `tg.devtools`.

TurboGears2 This is the actual framework. If you are writing an application which utilizes TurboGears2, then this is the package you need to add as a dependency in your `setup.py` file.

tg.devtools This package contains tools and dependencies to help you during your development process. It includes the Paste HTTP server, quickstart templates, and other tools. You will generally *not* list this as a dependency in your `setup.py` file, though you may have reason to do so. The application we will be developing for this book does not rely on it, and will not list it.

After activating your virtualenv, you only need to run one command:

```
$ pip install tg.devtools
```

That's it. Once it completes, you now have the TurboGears2 framework and development tools installed.

3.2 The GearBox Toolkit

The GearBox toolkit is a set of commands available since TurboGears 2.3 that replaced the paster command previously provided by pylons.

GearBox provides commands to create new full stack projects, serve PasteDeploy based applications, initialize them and their database, run migrations and start an interactive shell to work with them.

By default launching `gearbox` without any subcommand will start the interactive mode. This provides an interactive prompt where `gearbox` commands, system shell commands and python statements can be executed. If you have any doubt about what you can do simply run the `help` command to get a list of the commands available (running `help somecommand` will provide help for the given sub command).

To have a list of all the available commands simply run `gearbox --help`

3.2.1 QuickStart

The `gearbox quickstart` command creates a new full stack TurboGears application, just provide the name of your project to the command to create a new one:

```
$ gearbox quickstart myproject
```

The `quickstart` command provides a bunch of options to choose which template engine to use, which database engine to use various other options:

optional arguments:

```
-a, --auth          add authentication and authorization support
-n, --noauth       No authorization support
-m, --mako         default templates mako
-j, --jinja        default templates jinja
-k, --kajiki       default templates kajiki
-g, --geo          add GIS support
-p PACKAGE, --package PACKAGE
                   package name for the code
-s, --sqlalchemy   use SQLAlchemy as ORM
-i, --ming         use Ming as ORM
-x, --nosa         No SQLAlchemy
--disable-migrations
                   disable sqlalchemy-migrate model migrations
--enable-twl       use toscawidgets 1.x in place of 2.x version
--skip-tw         Disables ToscaWidgets
--noinput         no input (don't ask any questions)
```

3.2.2 Setup-App

The `gearbox setup-app` command runs the `websetup.setup_app` function of your project to initialize the database schema and data.

By default the `setup-app` command is run on the `development.ini` file, to change this provide a different one to the `--config` option:

```
$ gearbox setup-app -c production.ini
```

3.2.3 Serve

The `gearbox serve` command starts a PasteDeploy web application defined by the provided configuration file. By default the `development.ini` file is used, to change this provide a different one to the `--config` option:

```
$ gearbox serve -c production.ini --reload
```

The `serve` command provides a bunch of options to start the serve in daemon mode, automatically restart the application whenever the code changes and many more:

optional arguments:

```
-c CONFIG_FILE, --config CONFIG_FILE
                        application config file to read (default:
                        development.ini)
-n NAME, --app-name NAME
                        Load the named application (default main)
-s SERVER_TYPE, --server SERVER_TYPE
                        Use the named server.
--server-name SECTION_NAME
                        Use the named server as defined in the configuration
                        file (default: main)
--daemon
                        Run in daemon (background) mode
--pid-file FILENAME
                        Save PID to file (default to gearbox.pid if running in
                        daemon mode)
--reload
                        Use auto-restart file monitor
--reload-interval RELOAD_INTERVAL
                        Seconds between checking files (low number can cause
                        significant CPU usage)
--monitor-restart
                        Auto-restart server if it dies
--status
                        Show the status of the (presumably daemonized) server
--user USERNAME
                        Set the user (usually only possible when run as root)
--group GROUP
                        Set the group (usually only possible when run as root)
--stop-daemon
                        Stop a daemonized server (given a PID file, or default
                        gearbox.pid file)
```

Changing HTTP Server

`gearbox serve` will look for the `[server:main]` configuration section to choose which server to run and one which port and address to listen.

Any PasteDeploy compatible server can be used, by default the `egg:gearbox#wsgiref` one is used, which is single threaded and based on python wsgiref implementation.

This server is idea for debugging as being single threaded removes concurrency issues and keeps around request local data, but should never be used on production.

On production system you might want to use `egg:gearbox#cherrypy` or `egg:gearbox#gevent` servers which run the application on CherryPy and Gevent, it is also possible to use other servers like Waitress (`egg:waitress#main`) if available.

3.2.4 TGShell

The `gearbox tgshell` command will load a TurboGears application and start an interactive shell inside the application.

The application to load is defined by the configuration file, by default `development.ini` is used, to load a different application or under a different configuration provide a configuration file using the `--config` option:

```
$ gearbox tgshell -c production.ini
```

The `tgshell` command provides an already active fake request which makes possible to call functions that depend on `tg.request`, it is also provided an `app` object through which is possible to make requests:

```
$ gearbox tgshell
TurboGears2 Interactive Shell
Python 2.7.3 (default, Aug 1 2012, 05:14:39)
[GCC 4.6.3]

All objects from myapp.lib.base are available
Additional Objects:
wsgiapp - This project's WSGI App instance
app      - WebTest.TestApp wrapped around wsgiapp

>>> tg.request
<Request at 0x3c963d0 GET http://localhost/_test_vars>
>>> app.get('/data.json').body
'{"params": {}, "page": "data"}'
>>> model.DBSession.query(model.User).first()
<User: name=manager, email=manager@somedomain.com, display=Example manager>
```

3.3 RESTful Web Applications with TurboGears

If you are developing an application where you want to expose your database using a stateless API, `tg.RestController` might be for you. If you want to serve resources with multiple formats, and handle embedded resource lookup, you might find `RestController` useful. If you want to provide simple URLs which are consistent across all of the data shared and manipulated in your application, `RestController` is probably worth a look.

Unlike `TGController`, `RestController` provides a mechanism to access the request's method, not just the URL. If you are not familiar with how HTTP requests work, think for a minute about the difference between sending a form with GET and POST. Primarily, developers use POST to send data to modify the database. They use GET to retrieve data. These are HTTP methods.

Standard HTTP verbiage includes: GET, POST, PUT, and DELETE. `RestController` supports these, and also adds a few shortcuts for URL dispatch that makes displaying the data as forms and lists a little easier for the user. The API docs describe each of the supported controller functions in brief, so use that if you already understand REST and need a quick way to start using it, this document will be your guide. It is intended to provide a step-by-step example of how to implement REST using `RestController`.

To explain how RESTful works with TurboGears we are going to define a simple webservice that exposes a list of movies. `WebServices` are usually an ideal candidate for RESTful dispatch and so provide a simple and clean showcase of the feature.

Here is the Model used to develop this chapter:

```
from sqlalchemy import Column, Integer, String, Date, Text, ForeignKey, Table
from sqlalchemy.orm import relation

from moviedemo.model import DeclarativeBase, metadata

movie_directors_table = Table('movie_directors', metadata,
                              Column('movie_id', Integer, ForeignKey('movies.movie_id'), primary_key=True),
                              Column('director_id', Integer, ForeignKey('directors.director_id'), primary_key=True))

class Genre(DeclarativeBase):
    __tablename__ = "genres"

    genre_id = Column(Integer, primary_key=True)
    name = Column(String(100))

class Movie(DeclarativeBase):
    __tablename__ = "movies"

    movie_id = Column(Integer, primary_key=True)
    title = Column(String(100), nullable=False)
    description = Column(Text, nullable=True)
    genre_id = Column(Integer, ForeignKey('genres.genre_id'))
    genre = relation('Genre', backref='movies')
    release_date = Column(Date, nullable=True)

class Director(DeclarativeBase):
    __tablename__ = "directors"

    director_id = Column(Integer, primary_key=True)
    title = Column(String(100), nullable=False)
    movies = relation(Movie, secondary=movie_directors_table, backref="directors")
```

I am isolating Movies, Genres, and Directors for the purpose of understanding how objects might relate to one another in a RESTful context. For purposes of this demonstration, Movies can only have one Genre, but may be related to one or more Directors. Directors may be related to one or more Movies.

3.3.1 Listing Resources

Lets provide a simple listing of the movies in our database.

Our controller class is going to look like this:

```
from tg import RestController
from tg.decorators import with_trailing_slash

class MovieController(RestController):

    @expose('json')
    def get_all(self):
        movies = DBSession.query(Movie).all()
        return dict(movies=movies)
```

Supposing our MovieController is mounted with the name movies inside our RootController going to <http://localhost:8080/movies> will provide the list of our movies encoded in json format.

If you are looking for a way to fill some sample movies, just jump to <http://localhost:8080/admin> and create any data you need to make sure your controller is working as expected.

3.3.2 Creating New Items

We use the *post* method to define how we go about saving our movie to the database. This method gets called whenever the `http://localhost:8080/movies` url is accessed using a POST request:

```
from datetime import datetime

class MovieRestController(RestController):

    @expose('json')
    def post(self, title, description, directors=None, genre_id=None, release_date=None):
        if genre_id is not None:
            genre_id = int(genre_id)

        if directors is not None:
            if not isinstance(directors, list):
                directors = [directors]
            directors = [DBSession.query(Director).get(director) for director in directors]
        else:
            directors = []

        if release_date is not None:
            release_date = datetime.strptime(release_date, "%m/%d/%y")

        movie = Movie(title=title, description=description, release_date=release_date,
                      directors=directors, genre_id=genre_id)
        DBSession.add(movie)
        DBSession.flush()

        return dict(movie=movie)
```

If the insertion is successful we are going to receive back the newly created movie with its `movie_id`. The `DBSession.flush()` call is explicitly there to make SQLAlchemy get a `movie_id` for the newly inserted movie.

This will not be the case if the user enters some weird date format for “`release_date`” or doesn’t provide a title or description.

One way to counteract this problem is by writing a validator when the parameters don’t respect the expected format.

If you don’t know how to test this controller, check for browser extension to make POST requests. Most browser have one, for Google Chrome you can try [PostMan](#) which does a good job.

Validating The User’s Input

Before we add our record to the database, it is probably a good idea to validate the data so we can prompt the user if there are mistakes. `RestController` uses the same machinery that `TGControllers` use for validation. We use `FormEncode`’s validators to test that our fields are not empty, and that the `release_date` has correct formatting:

```
from tg import request, validate
from formencode.validators import NotEmpty, Int, DateConverter

@validate({'title':NotEmpty,
          'description':NotEmpty,
          'genre_id':Int(not_empty=True),
          'release_date':DateConverter(not_empty=True)})
@expose('json')
def post(self, **kw):
    if request.validation['errors']:
```

```
        return dict(errors=dict([(field, str(e)) for field, e in request.validation['errors'].items()])
#...proceed like before...
```

Note that the validation errors are stored in `request.validation`. This is done by the TG dispatch on a failed validation.

3.3.3 Getting one Item

Using the `get_one()` method, we can display one item from the database to the user:

```
@expose('json')
def get_one(self, movie_id):
    movie = DBSession.query(Movie).get(movie_id)
    return dict(movie=movie)
```

3.3.4 Updating an Existing Item

PUT is the method used for updating an existing record using REST. We can validate in the same manner as before:

```
@validate({'title':NotEmpty,
          'description':NotEmpty,
          'genre_id':Int(not_empty=True),
          'release_date':DateConverter(not_empty=True)})
@expose('json')
def put(self, movie_id, title, description, directors, genre_id, release_date, **kw):
    if request.validation['errors']:
        return dict(errors=dict([(field, str(e)) for field, e in request.validation['errors'].items()])

    movie = DBSession.query(Movie).get(movie_id)
    if not movie:
        return dict(errors={'movie':'Movie not found'})

    genre_id = int(genre_id)
    if not isinstance(directors, list):
        directors = [directors]
    directors = [DBSession.query(Director).get(director) for director in directors]

    movie.genre_id = genre_id
    movie.title=title
    movie.description = description
    movie.directors = directors
    movie.release_date = release_date

    return dict(movie=movie)
```

3.3.5 Deleting An Item From Our Resource

The work-horse of delete is attached to the `post_delete` method. Here we actually remove the record from the database, and then redirect back to the listing page:

```
@expose('json')
def post_delete(self, movie_id, **kw):
    movie = DBSession.query(Movie).get(movie_id)
    if not movie:
```

```

    return dict(errors={'movie': 'Movie not found'})

DBSession.delete(movie)
return dict(movie=movie.movie_id)

```

3.3.6 Nesting Resources With RestControllers

RestControllers expect nesting as any TG controller would, but it uses a different method of dispatch than regular TG Controllers. This is necessary when you need resources that are related to other resources. This can be a matter of perspective, or a hard-link which filters the results of the sub controller. For our example, we will use a nested controller to display all of the directors associated with a Movie.

The challenge for design of your RESTful interface is determining how to associate parts of the URL to the resource definition, and defining which parts of the URL are part of the dispatch.

To do this, RestController introspects the `get_one` method to determine how many bits of the URL to nip off and makes them available inside the `request.controller_state.routing_args` dictionary.

This is because you may have one or more identifiers to determine an object; for instance you might use lat/lon to define a location. Since our MovieController defines a `get_one` which takes a `movie_id` as a parameter, we have no work to do there.

All we have to do now is define our MovieDirectorController, and provide linkage into the MovieController to provide this functionality:

```

from tg import request

class MovieDirectorController(RestController):
    @expose('json')
    def get_all(self):
        movie_id = request.controller_state.routing_args.get('movie_id')
        movie = DBSession.query(Movie).get(movie_id)
        return dict(movie=movie, directors=movie.directors)

class MovieRestController(RestController):
    directors = MovieDirectorController()

    @expose('json')
    def get_one(self, movie_id):
        movie = DBSession.query(Movie).get(movie_id)
        return dict(movie=movie)

```

This example only defines the `get_all` function, I leave the other RESTful verbiage as an exercise for you to do.

One trick that I will explain, is how to use `_before` to get the related Movie object within all of your MovieDirectorController methods with a single define.

Here is what the Controller looks like with `_before` added in:

```

from tg import tmpl_context, request

class MovieDirectorController(RestController):

    def _before(self, *args, **kw):
        movie_id = request.controller_state.routing_args.get('movie_id')
        tmpl_context.movie = DBSession.query(Movie).get(movie_id)

    @with_trailing_slash

```

```
@expose('json')
def get_all(self):
    return dict(movie=tmpl_context.movie, directors=tmpl_context.movie.directors)
```

3.3.7 Non-RESTful Methods

Let's face it, REST is cool, but sometimes it doesn't meet our needs or time constraints. A good example of this is a case where you want an autocomplete dropdown in your "edit" form, but the resource that would provide the Json for this dropdown has not been fleshed out yet. As a hack, you might add a `field_dropdown()` method in your controller which sends back the json required to feed your form. RestController allows methods named outside of the boundaries of the default methods supported. In other words, it's just fine to include a method in your RestController that does not fit the REST HTML verbiage specification.

Supporting TGController's Inside RestController

Just as RestController supports obscure names for methods, it can handle nested TGController classes as well. When dispatch encounters a URL which maps to a non-RestController, it switches back to the normal TG dispatch. Simply said, you may include regular classes for dispatch within your RestController definition.

3.4 ObjectDispatch and TGController

The TGController is the basic controller class that provides an easy method for nesting of controller classes to map URL hierarchies. There are however a few methods which provide ways to implement custom dispatching and some entry points that will make easy for the developer to track the progress of request dispatch.

3.4.1 Dispatching Entry Points

Dispatching entry points are methods that get executed while the dispatch is moving forward, this permits to run custom code which is related to the controller we are dispatching and not a specific method itself:

```
class Controller(BaseController):
    def _before(self, *remainder, **params):
        # Executed before running any method of Controller

    def _after(self, *remainder, **params):
        # Executed after running any method of Controller

    def _visit(self, *remainder, **params):
        # Executed when visiting a controller during dispatch.
```

- **_before gets executed whenever the dispatch process** decides that the request has to be served by a method of the controller, before calling the method itself. It is executed before any method *requirement* specified through `@require` has been evaluated, but after the controller `allow_only` has been evaluated.
- **_after gets executed after the request has been dispatched** to one of the controller methods.
- **_visit gets executed whenever the controller is visited** during the dispatch process. Actual request target might be a subcontroller and not the controller itself. Might be called multiple times and gets executed before `allow_only` has been evaluated.

3.4.2 The Default Method

The developer may decide to provide a `_default` method within their controller which is called when the dispatch mechanism cannot find an appropriate method in your controllers to call. This `_default` method might look something like this:

```
class WikiController(BaseController):

    @expose('mytgapp.wiki.new')
    def _default(self, *args):
        """
        Return a page to prompt the user to create a new wiki page."""
        """
        return dict(new_page_slug=args)s
```

3.4.3 The Lookup Method

`_lookup` and `_default` are called in identical situations: when “normal” object traversal is not able to find an exposed method, it begins popping the stack of “not found” handlers. If the handler is a “`_default`” method, it is called with the rest of the path as positional parameters passed into the default method.

The not found handler stack can also contain “lookup” methods, which are different, as they are not actual controllers.

A lookup method takes as its argument the remaining path elements and returns an object (representing the next step in the traversal) and a (possibly modified) list of remaining path elements. So a blog might have controllers that look something like this:

```
class BlogController(BaseController):
    @expose()
    def _lookup(self, year, month, day, id, *remainder):
        dt = date(int(year), int(month), int(day))
        blog_entry = BlogEntryController(dt, int(id))
        return blog_entry, remainder

class BlogEntryController(object):
    def __init__(self, dt, id):
        self.entry = model.BlogEntry.get_by(date=dt, id=id)

    @expose(...)
    def index(self):
        ...

    @expose(...)
    def edit(self):
        ...

    @expose()
    def update(self):
        ....
```

So a URL request to `.../2007/6/28/0/edit` would map first to the `BlogController`’s `_lookup` method, which would lookup the date, instantiate a new `BlogEntryController` object (`blog_entry`), and pass that `blog_entry` object back to the object dispatcher, which uses the remainder to continue dispatch, finding the `edit` method. And of course the `edit` method would have access to `self.entry`, which was looked up and saved in the object along the way.

In other situations, you might have a several-layers-deep “`_lookup`” chain, e.g. for editing hierarchical data (`/client/1/project/2/task/3/edit`).

The benefit over “_default” handlers is that you *return* an object that acts as a sub-controller and continue traversing rather than *being* a controller and stopping traversal altogether. This allows you to use actual objects with data in your controllers.

Plus, it makes RESTful URLs much easier than they were in TurboGears 1.

3.4.4 Subclassing Controllers

When overriding a parent controller method you will usually have to expose it again and place any validation or event hook it previously had.

While this is possible, it is not the best way to add additional behavior to existing controllers. If they are implemented in an external library or application, you will have to look at the code of the library, see any template it exposed, any hook it registered and place them again.

If the library will change in any future release your code will probably stop working.

To avoid this behavior and the issues it raises since TurboGears 2.2 it is possible to subclass controllers inheriting the configuration the parent methods had.

The `inherit` parameter of the `tg.decorators.expose` decorator enables this behavior:

```
class OriginalController(TGController):
    @expose('mylib.templates.index')
    def index(self):
        return dict()

    @expose('mylib.templates.about')
    def about(self):
        return dict()

    @expose('json')
    def data(self):
        return {'v':5}

class MyCustomizedController(OriginalController):
    @expose(inherit=True)
    def index(self, *args, **kw):
        dosomething()
        return super(MyCustomizedController, self).index(*args, **kw)

    @expose('myapp.templates.newabout', inherit=True)
    def about(self):
        return super(MyCustomizedController, self).about(*args, **kw)

    def _before_render_data(remainder, params, output):
        output['child_value'] = 'CHILDVALUE'

    @expose(inherit=True)
    @before_render(_before_render_data)
    def data(self, *args, **kw):
        return super(MyCustomizedController, self).data(*args, **kw)
```

3.4.5 Mount Points and Dispatch

Since TurboGears 2.1.4 it is possible to ask for various informations about the request dispatchment and controllers mount points.

Those informations can be useful when writing controllers that you plan to reuse in multiple applications or mount points, making possible for example to generate all the urls knowing where they are mounted.

For statically mounted controllers the exposed informations are:

- The `mount_point` property of a controller. If statically mounted it will return where the controller is mounted. This is the url to call when you want to access that controller.
- The `mount_steps` property of a controller. If statically mounted it will return the complete list of parents of that controller.

In the case you are dispatching the request yourself, for example through a `_lookup` method, the `mount_point` and `mount_steps` informations won't be available. In this case you can rely on some other functions exposed by TG:

- The `tg.request.controller_state` object keeps track of all the steps provided to dispatch the request.
- The `tg.dispatched_controller()` method when called inside a request will return the last statically mounted controller. This can be useful to detect which controller finished the request dispatch using the `_lookup` method.

The application `RootController` can usually be retrieved from `tg.config['application_root_module'].RootController`

3.5 Database Schema Migrations

Since version 2.1.1 TurboGears has integrated migrations support for each new quickstarted project.

TurboGears 2.3 and newer rely on the [alembic](#) project to automate database schema migration.

3.5.1 Getting Started

TurboGears provides a `gearbox migrate` command to manage schema migration. You can run `gearbox migrate db_version` to see the current version of your schema:

```
$ gearbox migrate -c development.ini db_version
Context impl SQLiteImpl.
Will assume transactional DDL.
Current revision for sqlite:///tmp/migr/devdata.db: None
```

By default the database version is `None` until a migration is applied. The first time a migration is applied the `migrate_version` table is created. This table will keep the current version of your schema to track when applying migrations is required.

If you examine your database, you should be able to see schema version tracking table and check what it is the current version of your schema:

```
sqlite> .headers on
sqlite> select * from migrate_version;
version_num
4681af2393c8
```

This is exactly like running the `gearbox migrate db_version` command, both should tell you the same database version. In this case the reported version is `4681af2393c8`.

Integrating Migrations in the Development Process

With the database under version control and a repository for schema change scripts, you are ready to begin regular development. We will now walk through the process of creating, testing, and applying a change script for your current database schema. Repeat these steps as your data model evolves to keep your databases in sync with your model.

3.5.2 Creating migrations

The `gearbox migrate script` command will create an empty change script for you, automatically naming it and placing it in your repository:

```
$ gearbox migrate create 'Initial Schema'
```

The command will return by just printing the migrations repository where it is going to create the new script:

```
$ gearbox migrate create 'Initial Schema'
    Generating /tmp/migr/migration/versions/2a3f515bad0_initial_schema.py... done

$ ls migration/versions
2a3f515bad0_this_is_an_example.py
```

Edit the Script

Each change script provides an `upgrade` and `downgrade` method, and we implement those methods by creating and dropping the account table respectively:

```
revision = '2a3f515bad0'
down_revision = '4681af2393c8'

from alembic import op
import sqlalchemy as sa

def upgrade():
    op.create_table(
        'account',
        sa.Column('id', sa.Integer, primary_key=True),
        sa.Column('name', sa.String(50), nullable=False),
        sa.Column('description', sa.Unicode(200)),
    )

def downgrade():
    op.drop_table('account')
```

Test the Script

Anyone who has experienced a failed schema upgrade on a production database knows how uniquely uncomfortable that situation can be. Although testing a new change script is optional, it is clearly a good idea. After you execute the following test command, you will ideally be successful:

```
$ gearbox migrate test
Context impl SQLiteImpl.
Will assume transactional DDL.
Running upgrade 4681af2393c8 -> 2a3f515bad0
Context impl SQLiteImpl.
```

Will assume transactional DDL.
Running downgrade 2a3f515bad0 -> 4681af2393c8

If you receive an error while testing your script, one of two issues is probably the cause:

- There is a bug in the script
- You are testing a script that conflicts with the schema as it currently exists.

If there is a bug in your change script, you can fix the bug and rerun the test.

3.5.3 Applying migrations

The script is now ready to be deployed:

```
$ gearbox migrate upgrade
```

If your database is already at the most recent revision, the command will produce no output. If migrations are applied, you will see output similar to the following:

```
Context impl SQLiteImpl.  
Will assume transactional DDL.  
Running upgrade 4681af2393c8 -> 2a3f515bad0
```

3.5.4 Keeping your websetup on sync

Each time you create a new migration you should consider keeping your websetup in sync with it. For example if you create a new table inside a migration when you will run `gearbox setup-app` on a new database it will already have the new table as you probably declared it in your model too but the migrations version will be `None`. So trying to run any migration will probably crash due to the existing tables.

To prevent this your `websetup` script should always initialize the database in the same state where it would be after applying all the available migrations. To ensure this you will have to add at the end of the `websetup/schema.py` script a pool of commands to set the schema version to the last one:

```
import alembic.config, alembic.command  
alembic_cfg = alembic.config.Config()  
alembic_cfg.set_main_option("script_location", "migration")  
alembic_cfg.set_main_option("sqlalchemy.url", config['sqlalchemy.url'])  
alembic.command.stamp(alembic_cfg, "head")
```

3.5.5 Downgrading your schema

There are some cases in which downgrading your schema might be required. In those cases you can perform the `gearbox migrate downgrade` command:

```
$ gearbox migrate downgrade  
Context impl SQLiteImpl.  
Will assume transactional DDL.  
Running downgrade 2a3f515bad0 -> 4681af2393c8
```

3.6 Testing with TurboGears

TurboGears quickstart command already provides a fully working test suite for your newly quickstarted app.

3.6.1 Running Tests

The only required dependency to run the testsuite is the `nose` package, which can be installed with:

```
$ pip install nose
```

Other dependencies used by your tests will automatically be installed when the test suite is run through the `install_requires` and `testpkgs` variables in `setup.py`, so if your application requires any dependency specific for testing just make sure is listed there.

To actually run the test suite you can run:

```
$ python setup.py nosetests
```

This will install all the required dependencies and will run the tests. You should get an output similar to:

```
Wrong password keeps user_name in login form ... ok
Anonymous users are forced to login ... ok
Logouts must work correctly ... ok
Voluntary logins must work correctly ... ok
The data display demo works with HTML ... ok
The data display demo works with JSON ... ok
Displaying the wsgi environ works ... ok
The front page is working properly ... ok
Anonymous users must not access the secure controller ... ok
The editor cannot access the secure controller ... ok
The manager can access the secure controller ... ok
plain.tests.functional.test_root.TestWithContextController.test_i18n ... ok
Model objects can be created ... ok
Model objects can be queried ... ok
Model objects can be created ... ok
Model objects can be queried ... ok
Model objects can be created ... ok
Users should be fetchable by their email addresses ... ok
User objects should have no permission by default. ... ok
The obj constructor must set the email right ... ok
The obj constructor must set the user name right ... ok
Model objects can be queried ... ok
```

```
-----
Ran 22 tests in 11.931s
```

```
OK
```

Those are the default tests that TurboGears generated for you.

Tests Collection

When running the `nosetests` command nose will look for any `tests` directory inside your application package.

Nose itself will look in all files whose name starts with `test_[something].py` for all the classes which name starts with `Test[Something]` and will consider them as Test Cases, for each method inside the test case whose name starts with `test_[something]` they will be treated as Test Units.

3.6.2 Writing Tests

When quickstarting an application you will notice that there is a `tests` package inside it. This package is provided by TurboGears itself and contains the fixture already creates a `TestApp` instance for you and loads configuration from

`test.ini` instead of `development.ini`.

The `TestApp` which is available inside *Test Cases* as `self.app` is an object with methods that emulate HTTP requests: `.get`, `.post`, `.put` and so on and is able to understand both **html** and **json** responses.

Take note that `test.ini` actually inherits from `development.ini` and just overwrites some options. For example for tests by default a `sqlalchemy.url = sqlite:///memory:` is used which forces SQLAlchemy to use an in memory database instead of a real one, so that it is created and discarded when the test suite is run without requiring you to use a real database.

All your application tests that call a web page should inherit from `tests.TestController` which ensure:

- For each test unit the database is created and initialized by calling `setup-app`.
- For each test unit the `self.app` object is provided which is a `TestApp` instance of your TurboGears2 application loaded from `test.ini`.
- After each test unit the database is deleted.
- After each test unit the SQLAlchemy session is cleared.

For a sample test see `tests/functional/test_root`:

```
from nose.tools import ok_
from testapp.tests import TestController

class TestRootController(TestController):
    """Tests for the method in the root controller."""

    def test_index(self):
        response = self.app.get('/')
        msg = 'TurboGears 2 is rapid web application development toolkit '\
            'designed to make your life easier.'
        ok_(msg in response)

    def test_environ(self):
        response = self.app.get('/environ.html')
        ok_('The keys in the environment are:' in response)
```

Simulating Authentication Requests

To simulate authentication you can just pass to the `.get`, `.post` and so on methods an `extra_environ` parameter (which is used to add WSGI environ values available in `tg.request.environ`) named `REMOTE_USER`.

For example if you want to behave like you are logged as the editor user you just pass:

```
def test_secc_with_editor(self):
    environ = {'REMOTE_USER': 'editor'}
    self.app.get('/secc', extra_environ=environ, status=403)
```

The previous test will check that when user is logged as *editor* a `403` error is returned from the `/secc` url instead of the `401` which is returned when user is not logged at all.

Checking HTML Responses

Within the tests it is also possible to check complex HTML structures if the `pyquery` module is installed.

To install `pyquery` just add it to your `testpkgs` in `setup.py` so that it will be automatically installed when running the test suite.

PyQuery is a python module that works like jQuery and permits easy traversing of the DOM:

```
from testapp.tests import TestController

class TestHelloWorldApp(TestController):
    """Tests an app that returns a simple HTML response with:

        <html>
            <head>
                <title>Hello to You</title>
            </head>
            <body>
                <h1>Hello World</h1>
            </body>
        </html>
    """

    def test_hello_world(self):
        res = self.app.get('/')
        assert res.pyquery('h1').text() == 'Hello World'

    def test_title(self):
        res = self.app.get('/')
        assert res.pyquery('title').text() == 'Hello to You'
```

For pyquery documentation please rely on <https://pythonhosted.org/pyquery/>

Submitting Forms

The TestApp permits also to easily fill and submit forms, this can be used to test features that require submission of form values:

```
from testapp.tests import TestController

class TestFormApp(TestController):
    """Tests an app that contains a simple HTML form with:

        <form id="form1" action="/submit" method="POST">
            <input type="text" name="value"/>
        </form>

        That submits to:

        @expose('json')
        def submit(self, value=None, **kwargs):
            return dict(value=value)
    """

    def test_form_submission(self):
        page = self.app.get('/')

        form = page.forms['form1']
        form['value'] = 'prova'

        res = form.submit()
        assert res.json['value'] == 'prova', res
```

The *form* itself is identified by its **id**, so the `page.forms['form1']` works as the form has `id="form1"`.

Testing Outside Controllers

There might be cases when you are required to test something outside a controller, this is common with validators or utility methods.

In those cases you can inherit from `tests.TestController` as usual, and you will probably won't use the `self.app` object. Unless you are required to have a request in place during your test.

This might be the case if your utility function or class uses TurboGears features that depend on a request like `tg.url`, `tg.i18n.gettext` and so on...

Since version 2.3.5 you can call a special URL in tests to initialize a fake request context which will be used for the whole test lifetime:

```
from testapp.tests import TestController

class TestWithContextController(TestController):
    def test_i18n(self):
        self.app.get('/_test_vars') # Initialize a fake request

        hello = gettext('hello')
        assert hello == 'hello', hello
```

Make sure you reset the request context after using `/_test_vars` otherwise you might end up with a messy environment as you have left behind the globally registered objects. It is a good practice to perform another another request to properly reset the global object status at the end of the test unit:

```
from testapp.tests import TestController

class TestWithContextController(TestController):
    def tearDown(self):
        self.app.get('/_del_test_vars', status=404) # Reset fake request
        super(TestWithContextController, self).tearDown()

    def test_i18n(self):
        self.app.get('/_test_vars') # Initialize a fake request

        hello = gettext('hello')
        assert hello == 'hello', hello
```

3.6.3 Coverage

Coverage is the process of identifying all the paths of execution that the Test Suite is not checking. Aiming at 100% code coverage means that we are sure that our tests pass through all branches in our code and all the code we wrote has been run at least once.

Note that Coverage is able to guarantee that we checked everything we wrote, it is not able to measure code that we should have written but didn't. Missing errors handling won't be detected in coverage but it is a fairly reliable tool to ensure that everything your wrote has been checked at least once.

By default coverage reporting is disabled in turbogears test suite, but it can easily be turned on by changing `with-coverage` option in `setup.cfg`:

```
[nosetests]
verbosity = 2
detailed-errors = 1
with-coverage = true # CHANGED TO true TO ENABLE COVERAGE
cover-erase = true
cover-package = plain
```

When coverage is enabled, after the tests results, you will get the coverage report:

```
..
Name          Stmts  Miss  Cover   Missing
-----
_opnums         13     8    38%   2-3, 6-9, 12, 16-17
-----
Ran 2 tests in 0.002s

OK
```

3.7 TurboGears2 Configuration

3.7.1 TurboGears 2 Configuration

TurboGears 2 provides a configuration system that attempts to be both extremely flexible for power users and very simple to use for standard projects.

Overview

The application configuration is separated from the deployment specific information. In TurboGears 2.3.5 there is a config module, containing several configuration specific python files – these are done in python (not as INI files), because they actually setup the TurboGears 2.3.5 application and its associated WSGI middleware. Python provides an incredibly flexible config system with all kinds of tools to keep you from having to repeat yourself. But it comes with some significant drawbacks, python is more complex than INI, and is less declarative so can be less obvious.

But we believe these drawbacks are more than overcome by the power and flexibility of python based configuration for the app because these files are intended to be edited only by application developers, not by those deploying the application. We've also worked hard to create an environment that is generally declarative.

At the same time the deployment level configuration is done in simple .ini files, in order to make it totally declarative, and easy for deployers who may not be python programmers.

Configuration in the INI files

A TurboGears quickstarted project will contain a couple of .ini files which are used to define what WSGI app ought to be run, and to store end-user created configuration values, which is just another way of saying that the .ini files should contain *deployment specific* options.

By default TurboGears provides a `development.ini`, `test.ini`, and `production.ini` files. These are standard ini file formats.

These files are standard INI files, as used by PasteDeploy. The individual sections are marked off with `[]`'s.

See also:

Configuration file format **and options** are described in great detail in the [Paste Deploy documentation](#).

If want to add some configuration option (let's say an administrator's email) here is how you would do so. First you would edit your `development.ini` file and go to the end of the `[app:main]` section.

You can then choose a sensible name for your configuration key and add it to the section:

```
mail.from.administrator = someemail@somedomain.com
```

This would make sure this variable is now part of the configuration and can be accessed from anywhere in your code. For example let's imagine that you wanted to get this config option from a controller's code:

```
import tg
admin_emailfrom = tg.config.get('mail.from.administrator', 'notconfigured@nodomain.com')
```

If the person who deployed your application forgot to add the variable to his config file he would get the default value provided as the second argument of the `get()` call.

Warning: If you set a value like `enable_subsystem = false`, it will be loaded into python as the string 'false' which if used in a conditional will give you a very wrong result

The correct way of loading boolean values for your use is

```
from paste.deploy.converters import asbool
if asbool(config['enable_subsystem']):
    ... sub systems is enabled...
```

Configuration Milestones

Since TurboGears 2.3 the configuration process got divided in various milestones, each of those milestones is bound to an advancement in the framework setup process.

Whenever a milestone is reached all the registered callbacks are fired and the configuration process can continue. If the milestone is already passed when a callback is registered, the callback gets instantly fired.

Note: The `tg.config` object is available at import time but until the configuration file is parsed, it only contains the system defaults. If you need to perform startup time setup based on the supplied configuration, you should do so in a milestone.

Milestones are available through the `tg.configuration.milestones` module, the currently provided milestones are:

- **`milestones.config_ready`** - Configuration file has been loaded and is available in `tg.config`
- **`milestones.renderers_ready`** - Renderers have been registered and all of them are available
- **`milestones.environment_loaded`** - Full environment have been loaded but application has not been created yet.

Registering an action to be executed whenever a milestone is reach can be done using `tg.configuration.milestones._ConfigMilestoneTracker.register()` method of each milestone. The registered action takes no parameters.

Milestones are much like *Hooks* but they are only related to the configuration process. The major difference is that *while an hook can fire multiple times a milestone can be reached only once.*

Milestones and Hooks order of execution

The order of execution of the milestones and hooks provided during the application startup process is:

- `milestones.config_ready`
- *startup Hook*
- `milestones.renderers_ready`
- `milestones.environment_loaded`
- *before_config Hook*
- *after_config Hook*

The config module

Tip: A good indicator of whether an option should be set in the `config` directory code vs. the configuration file is whether or not the option is necessary for the functioning of the application. If the application won't function without the setting, it belongs in the appropriate *config/* directory file. If the option should be changed depending on deployment, it belongs in the ini files.

Our hope is that 90% of applications don't need to edit any of the config module files, but for those who do, the most common file to change is `app_config.py`:

```
from tg.configuration import AppConfig
import wiki20
from wiki20 import model
from wiki20.lib import app_globals, helpers

base_config = AppConfig()
base_config.renderers = []

base_config.package = wiki20

#Set the default renderer
base_config.default_renderer = 'genshi'
base_config.renderers.append('genshi')

#Configure the base SQLAlchemy Setup
base_config.use_sqlalchemy = True
base_config.model = wiki20.model
base_config.DBSession = wiki20.model.DBSession
```

`app_cfg.py` exists primarily so that `middleware.py` and `environment.py` can import and use the `base_config` object.

The `base_config` object is an `AppConfig()` instance which allows you to access its attributes like a normal object, or like a standard python dictionary.

One of the reasons for this is that `AppConfig()` provides some defaults in its `__init__`. But equally important it provides us with several methods that work on the config values to produce the two functions that set up your TurboGears app.

We've taken care to make sure that the entire setup of the TurboGears 2.3.5 framework is done in code which you as the application developer control. You can easily customize it to your needs. If the standard config options we provide don't do what you need, you can subclass and override `AppConfig` to get exactly the setup you want.

The `base_config` object that is created in `app_cfg.py` should be used to set whatever configuration values that belong to the application itself and are required for all instances of this app, as distinct from the configuration values that you set in the `development.ini` or `production.ini` files that are intended to be editable by those who deploy the app.

As part of the app loading process the `base_config` object will be merged in with the config values from the `.ini` file you're using to launch your app, and placed in `tg.config`.

As we mentioned previously, in addition to the attributes on the `base_config` object there are a number of methods which are used to setup the environment for your application, and to create the actual TurboGears WSGI application, and all the middleware you need.

You can override `base_config`'s methods to further customize your application's WSGI stack, for various advanced use cases, like adding custom middleware at arbitrary points in the WSGI pipeline, or doing some unanticipated (by us) application environment manipulation.

And we'll look at the details of how that all works in the advanced configuration section of this document.

Configuring your application

Here's are some of the more general purpose configuration attributes:

Configuration Attributes

The configuration object has a number of attributes that automate the majority of what you need to do with the config object. These shortcuts eliminate the need to provide your own setup methods for configuring your TurboGears application.

Mimetypes By default, only `json/application` and `text/html` are defined mimetypes. If you would like to use additional mime-types you must register them with your application's config. You can accomplish this by adding the following code your your `app_cfg.py` file:

```
base_config.mimetype_lookup = { '.ext': 'my-mimetype' }
```

Static Files `base_config.serve_static` – automatically set to `True` for you. Set to `False` if you have set up apache, or nginx (or some other server) to handles static files.

Request Extensions `base_config.disable_request_extensions` – by default this is false. This means that TG will take the request, and strip anything off the end of the last element in the URL that follows `”.”`. It will then take this information, and assign an appropriate mime-type and store the data in the `tg.request.response_type` and `tg.request.response_ext` variables. By enabling this flag, you disable this behavior, rendering TG unable to determine the mime-type that the user is requesting automatically.

Stand Alone `base_config.stand_alone` – set this to `False` if you don't want error handling, HTTP status code error pages, etc. This is intended for the case where you're embedding the TG app in some other WSGI app which handles these things for you.

Cookie Secret The `beaker.session.secret` key of the `base_config` object contains the secret used to store user sessions. TurboGears automatically generates a random secret for you when you create a project. If an attacker gets his hands on this key, he will be able to forge a valid session an use your application at though he was logged in. In the event of a security breach, you can change this key to invalidate all user sessions.

Authentication Character Set Set `base_config.sa_auth.charset` to define the character encoding for your user's login. This is especially important if you expect your users to have non-ascii usernames and passwords. To set it to utf-8, your add this to your `app_config.py` file.:

```
base_config.sa_auth.charset = 'utf-8'
```

Advanced Configuration

Sometimes you need to go beyond the basics of setting configuration options. We've created a number of methods that you can use to override the way that particular pieces of the TurboGears 2.3.5 stack are configured. The basic way you override the configuration within `app.cfg` looks something like this:

```
from tg.configuration import AppConfig
from tw2.core.middleware import TwMiddleware

class MyAppConfig(AppConfig):

    def add_tosca2_middleware(self, app):

        app = TwMiddleware(app,
            default_engine=self.default_renderer,
            translator=ugettext,
            auto_reload_templates = False
        )

    return app

base_config = MyAppConfig()

# modify base_config parameters below
```

The above example shows how one would go about overriding the `ToscaWidgets2` middleware. See the `AppConfig` for more ideas on how you could modify your own custom config

3.7.2 SQLAlchemy and Transaction Config Settings

Table of Contents

- [SQLAlchemy and Transaction Config Settings](#)
 - [AppConfig Method Overrides](#)

Though the majority of folks will use TurboGears with SQLAlchemy, there are those who have interest in running the full stack of TG with a non-relational database like [MongoDB](#) or [CouchDB](#). There are a few settings that allow this, the most pertinent is: `use_sqlalchemy`:

`base_config.use_sqlalchemy` – Set to `False` to turn off sqlalchemy support

TurboGears takes advantage of `repoze`'s transaction manager software. Basically, the transaction manager wraps each of your controller methods, and should a method fail, the transaction will roll back. If you utilize the transaction manager, then the result of a successful method call results in a commit to the database. If the controller method does not utilize the database, there is no database interaction performed. What this means is that you never have to worry about committing, or rolling back when controller code fails, TG handles this for you automatically.

`base_config.use_transaction_manager` – Set to `False` to turn off the Transaction Manager and handle transactions yourself.

AppConfig Method Overrides

`AppConfig.setup_sqlalchemy()`
Setup SQLAlchemy database engine.

The most common reason for modifying this method is to add multiple database support. To do this you might modify your `app_cfg.py` file in the following manner:

```
from tg.configuration import AppConfig, config
from myapp.model import init_model

# add this before base_config =
class MultiDBAppConfig(AppConfig):
    def setup_sqlalchemy(self):
        '''Setup SQLAlchemy database engine(s)'''
        from sqlalchemy import engine_from_config
        engine1 = engine_from_config(config, 'sqlalchemy.first.')
        engine2 = engine_from_config(config, 'sqlalchemy.second.')
        # engine1 should be assigned to sa_engine as well as your first engine's name
        config['tg.app_globals'].sa_engine = engine1
        config['tg.app_globals'].sa_engine_first = engine1
        config['tg.app_globals'].sa_engine_second = engine2
        # Pass the engines to init_model, to be able to introspect tables
        init_model(engine1, engine2)

#base_config = AppConfig()
base_config = MultiDBAppConfig()
```

This will pull the config settings from your `.ini` files to create the necessary engines for use within your application. Make sure you have a look at [Using Multiple Databases In TurboGears](#) for more information.

`AppConfig.add_tm_middleware(app)`
Set up the transaction management middleware.

To abort a transaction inside a TG2 app:

```
import transaction
transaction.doom()
```

By default http error responses also roll back transactions, but this behavior can be overridden by overriding `base_config['tm.commit_veto']`.

3.7.3 Template Rendering Config Settings

Status Official

Table of Contents

- Template Rendering Config Settings
 - Configuration Attributes
 - Making a module available to all Genshi templates
 - Overriding AppConfig Rendering Methods

The most common configuration change you'll likely want to make here is to add a second template engine or change the template engine used by your project.

By default TurboGears sets up the Genshi engine, but we also provide out of the box support for Mako and Jinja. To tell TG to prepare these templating engines for you all you need to do is install the package and append 'mako' or 'jinja' to the renderer's list here in `app_config`.

To change the default renderer to something other than Genshi, just set the `default_renderer` to the name of the rendering engine. So, to add Mako to the list of renderers to prepare, and set it to be the default, this is all you'd have to do:

```
base_config.default_renderer = 'mako'
base_config.renderers.append('mako')
```

Configuration Attributes

`base_config.default_renderer` – set to the name of the default render function you want to use.

`base_config.renderers` – This is a list of rendering engines that ought to be prepared for use in the app. To make it available in your application you must specify here the name of the engine you want to use.

TG provides built-in renderers for: 'genshi', 'mako', 'jinja', 'json' and 'jsonp'.

In 2.3.2 and newer versions, If you would like to add additional renderers, you can add it to the renderers list, and then register a rendering engine factory through the `tg.configuration.AppConfig.register_rendering_engine()` method.

`base_config.use_dotted_templatenames` – Generally you will not want to change this. But if you want to use the standard genshi/mako/jinja file system based template search paths, set this to *False*. The main advantage of dotted template names is that it's very easy to store template files in zipped eggs, but if you're not using packaged TurboGears 2.3.5 app components there are some advantages to the search path syntax.

Making a module available to all Genshi templates

Sometimes you want to expose an entire module to all of the templates in your templates directory. Perhaps you have a form library you like to use, or a png-txt renderer that you want to wrap with `<pre>`. This is possible in TG.

First, we must modify our `app_cfg.py` so that you can share your link across all templates:

```
base_config.variable_provider = helpers.add_global_tmpl_vars
```

Next, you want to modify the `lib/helpers.py` module of your application to include the newly added `add_global_tmpl_vars` method:

```
import mymodule

def add_global_tmpl_vars():
    return dict(mymodule=mymodule)
```

That's pretty much it, you should have access to `mymodule` in every template now.

Overriding AppConfig Rendering Methods

Please have a look at `tg.configuration.AppConfig.register_rendering_engine()` for information on how to setup custom rendering engines.

3.8 Authentication in TurboGears 2 applications

This document describes how `repoze.who` is integrated into TurboGears and how you make get started with it. For more information, you may want to check `repoze.who`'s website.

`repoze.who` is a powerful and extensible authentication package for arbitrary WSGI applications. By default TurboGears2 configures it to log using a form and retrieving the user information through the `user_name` field of the `User` class. This is made possible by the `authenticator plugin` that TurboGears2 uses by default which asks `base_config.sa_auth.authmetadata` to authenticate the user against given login and password.

3.8.1 How it works in TurboGears

The authentication layer it's a WSGI middleware which is able to authenticate the user through the method you want (e.g., LDAP or HTTP authentication), "remember" the user in future requests and log the user out.

You can customize the interaction with the user through four kinds of *plugins*, sorted by the order in which they are run on each request:

- An `identifier plugin`, with no action required on the user's side, is able to tell whether it's possible to authenticate the user (e.g., if it finds HTTP Authentication headers in the HTTP request). If so, it will extract the data required for the authentication (e.g., username and password, or a session cookie). There may be many identifiers and `repoze.who` will run each of them until one finds the required data to authenticate the user.
- If at least one of the identifiers could find data necessary to authenticate the current user, then an `authenticator plugin` will try to use the extracted data to authenticate the user. There may be many authenticators and `repoze.who` will run each of them until one authenticates the user.
- When the user tries to access a protected area or the login page, a `challenger plugin` will come up to request an action from the user (e.g., enter a user name and password and then submit the form). The user's response will start another request on the application, which should be caught by an *identifier* to extract the login data and then such data will be used by the *authenticator*.
- For authenticated users, `repoze.who` provides the ability to load related data (e.g., real name, email) in the WSGI environment so that it can be easily used in the application. Such a functionality is provided by so-called metadata provider plugins. There may be many metadata providers and `repoze.who` will run them all.

When `repoze.who` needs to store data about the authenticated user in the WSGI environment, it uses its `repoze.who.identity` key, which can be accessed using the code below:

```
from tg import request

# The authenticated user's data kept by repoze.who:
identity = request.environ.get('repoze.who.identity')
```

Such a value is a dictionary and is often called "the identity dict". It will only be defined if the current user has been authenticated.

There is a short-cut to the code above in the WSGI `request`, which will be defined in `{yourproject}.lib.base.BaseController` if you enabled authentication and authorization when you created the project.

For example, to check whether the user has been authenticated you may use:

```
# ...
from tg import request
# ...
```

```
if request.identity:
    flash('You are authenticated!')
```

`request.identity` will equal to `None` if the user has not been authenticated.

Likewise, this short-cut is also set in the template context as `tg.identity`.

The `username` will be available in `identity['repoze.who.userid']` (or `request.identity['repoze.who.userid']`, depending on the method you select).

The FastFormPlugin

By default, TurboGears 2.3.5 configures `repoze.who` to use `tg.configuration.auth.fastform.FastFormPlugin` as the first identifier and challenger – using `/login` as the relative URL that will display the login form, `/login_handler` as the relative URL where the form will be sent and `/logout_handler` as the relative URL where the user will be logged out. The so-called rememberer of such identifier will be an instance of `repoze.who.plugins.cookie.AuthTktCookiePlugin`.

All these settings can be customized through the `config.app_cfg.base_config.sa_auth` options in your project. Identifiers, Authenticators and Challengers can be overridden providing a different list for each of them as:

```
base_config.sa_auth['identifiers'] = [('myidentifier', myidentifier)]
```

You don't have to use `repoze.who` directly either, unless you decide not to use it the way TurboGears configures it.

3.8.2 Customizing authentication and authorization

It's very easy for you to customize authentication and identification settings in `repoze.who` from `{yourproject}.config.app_cfg.base_config.sa_auth`.

Customizing how user informations, groups and permissions are retrieved

TurboGears provides an easy shortcut to customize how your authorization data is retrieved without having to face the complexity of the underlying authentication layer. This is performed by the `TGAuthMetadata` object which is configured in your project `config.app_cfg.base_config`.

This object provides three methods which have to return respectively the user, its groups and its permissions. You can freely change them as you wish as they are part of your own application behavior.

Advanced Customizations

For more advanced customizations or to use `repoze` plugins to implement different forms of authentication you can freely customize the whole authentication layer using through the `{yourproject}.config.app_cfg.base_config.sa_auth` options.

The available directives are all optional:

- **form_plugin:** This is a replacement for the **FriendlyForm** plugin and will be always used as a challenger. If `form_identifies` option is `True` it will also be appended to the list of identifiers.
- **identifiers:** A custom list of **repoze.who** identifiers. By default it contains the `form_plugin` and the `AuthTktCookiePlugin`.
- **challengers:** A custom list of **repoze.who** challengers. The `form_plugin` is always appended to this list, so if you have only one challenger you will want to change the `form_plugin` instead of overriding this list.

- **authmetadata:** This is the object that TG will use to fetch authorization metadata. Changing the auth-metadata object you will be able to change how TurboGears fetches your user data, groups and permissions. Using authmetadata a new `repoze.who` metadata provider is created.
- **mdproviders:** This is a list of `repoze.who` metadata providers. If `authmetadata` is not `None` a metadata provider based on it will always be appended to the `mdproviders`.

Customizing the model structure assumed by the quickstart

Your auth-related model doesn't *have to* be like the default one, where the class for your users, groups and permissions are, respectively, `User`, `Group` and `Permission`, and your users' user name is available in `User.user_name`. What if you prefer `Member` and `Team` instead of `User` and `Group`, respectively?

First of all we need to inform the authentication layer that our user is stored in a different class. This makes `repoze.who` know where to look for the user to check its password:

```
# what is the class you want to use to search for users in the database
base_config.sa_auth.user_class = model.Member
```

Then we have to tell out `authmetadata` how to retrieve the user, its groups and permissions:

```
from tg.configuration.auth import TGAAuthMetadata

#This tells to TurboGears how to retrieve the data for your user
class ApplicationAuthMetadata(TGAAuthMetadata):
    def __init__(self, sa_auth):
        self.sa_auth = sa_auth

    def authenticate(self, environ, identity):
        user = self.sa_auth.db.session.query(self.sa_auth.user_class).filter_by(user_name=identity['login'])
        if user and user.validate_password(identity['password']):
            return identity['login']

    def get_user(self, identity, userid):
        return self.sa_auth.user_class.query.get(user_name=userid)

    def get_groups(self, identity, userid):
        return [team.team_name for team in identity['user'].teams]

    def get_permissions(self, identity, userid):
        return [p.permission_name for p in identity['user'].permissions]

base_config.sa_auth.authmetadata = ApplicationAuthMetadata(base_config.sa_auth)
```

Now our application is able to fetch the user from the `Member` table and its groups from the `Team` table. Using `TGAAuthMetadata` makes also possible to introduce a caching layer to avoid performing too many queries to fetch the authentication data for each request.

BasicAuth Example

The following is an example of an advanced authentication stack customization to use browser basic authentication instead of form based authentication.

Declaring a Custom Authentication Backend

First required step is to declare that we are going to use a custom authentication backend:

```
base_config.auth_backend = 'htpasswd'
```

When this is valued to `ming` or `sqlalchemy` TurboGears will configure a default authentication stack based on users stored on the according database, if `auth_backend` is `None` the whole stack will be disabled.

Then we must remove all the simple authentication options, deleting all the `basic_config.sa_auth` from `app_cfg.py` is usually enough. Leaving unexpected options behind (options our authentication stack doesn't use) might lead to a crash on application startup.

Using HTPasswd file for users

Next step is storing our users inside an `htpasswd` file, this can be achieved by using the `HTPasswdPlugin` authenticator:

```
from repoze.who.plugins.htpasswd import HTPasswdPlugin, plain_check
base_config.sa_auth.authenticators = [('htpasswd', HTPasswdPlugin('./htpasswd', plain_check))]
```

This will make TurboGears load users from an `htpasswd` file inside the directory we are starting the application from. The `plain_check` function is the one used to decode password stored inside the `htpasswd` file. In this case passwords are expected to be in plain text in the form:

```
manager:managepass
```

Challenging and Identifying users with BasicAuth

Now that we are correctly able to authenticate users from an `htpasswd` file, we need to use `BasicAuth` for identifying returning users:

```
from repoze.who.plugins.basicauth import BasicAuthPlugin

base_auth = BasicAuthPlugin('MyTGApp')
base_config.sa_auth.identifiers = [('basicauth', base_auth)]
```

This will correctly identify users that are already logged using `BasicAuth`, but we are still sending users to login form to perform the actual login.

As `BasicAuth` requires the login to be performed through the browser we must disable the login form and set the basic auth plugin as a challenger:

```
# Disable the login form, it won't work anyway as the credentials
# for basic auth must be provided through the browser itself
base_config.sa_auth.form_identifies = False

# Use BasicAuth plugin to ask user for credentials, this will replace
# the whole login form.
base_config.sa_auth.challengers = [('basicauth', base_auth)]
```

Providing User Data

The previous steps are focused on providing a working authentication layer, but we will need to also identify the authenticated user so that also `request.identity` and the authorization layer can work as expected.

This is achieved through the `authmetadata` option, which tells TurboGears how to retrieve the user and it's informations. In this case as we don't have a database of users we will just provide a simple user with only `display_name`

and `user_name` so that most things can work. For manager user we will also provide the `managers` group so that user can access the TurboGears admin:

```
from tg.configuration.auth import TGAAuthMetadata

class ApplicationAuthMetadata(TGAAuthMetadata):
    def __init__(self, sa_auth):
        self.sa_auth = sa_auth

    def get_user(self, identity, userid):
        # As we use htpasswd for authentication
        # we cannot lookup the user in a database,
        # so just return a fake user object
        from tg.util import Bunch
        return Bunch(display_name=userid, user_name=userid)

    def get_groups(self, identity, userid):
        # If the user is manager we give him the
        # managers group, otherwise no groups
        if userid == 'manager':
            return ['managers']
        else:
            return []

    def get_permissions(self, identity, userid):
        return []
```

```
base_config.sa_auth.authmetadata = ApplicationAuthMetadata(base_config.sa_auth)
```

Removing Login Form

As the whole authentication is now performed through BasicAuth the login form is now unused, so probably want to remove the login form related urls which are now unused:

- `/login`
- `/post_login`
- `/post_logout`

3.8.3 Disabling authentication and authorization

If you need more flexibility than that provided by the quickstart, or you are not going to use `repoze.who`, you should prevent TurboGears from dealing with authentication/authorization by removing (or commenting) the following line from `{yourproject}.config.app_cfg`:

```
base_config.auth_backend = '{whatever you find here}'
```

Then you may also want to delete those settings like `base_config.sa_auth.*` – they'll be ignored.

3.9 Hooks and Wrappers

TurboGears defines three ways to plug behaviors inside existing applications and plugins: Hooks, Controller Wrappers and Application Wrappers.

Hooks work in an event registration and notification manner, they permit to emit events and notify registered listeners which are able to perform actions depending on the event itself.

Controller Wrappers sits between TurboGears and the controller body code, they permit to extend controllers code much like a decorator, but can be attached to third party controllers or application wide to any controller.

Application Wrappers are much like WSGI middlewares but behave and work in the TurboGears context, so they receive the TurboGears context and Request instead of the WSGI environ and are expected to return a `webob.Response` object back.

3.9.1 Hooks

TurboGears allows you to attach callables to a wide set of events. Most of those are available as both controller events and system wide events.

To register a system wide even you can use the `register` method of the `tg.hooks` object. As some hooks require being registered before the application is running, it's common practice to register them in your `app_cfg.py` file:

```
def on_startup():
    print 'hello, startup world'

def on_shutdown():
    print 'hello, shutdown world'

def before_render(remainder, params, output):
    print 'system wide before render'

# ... (base_config init code)
tg.hooks.register('startup', on_startup)
tg.hooks.register('shutdown', on_shutdown)
tg.hooks.register('before_render', before_render)
```

To register controller based hooks you can use the event decorators:

```
from tg.decorators import before_render

def before_render_cb(remainder, params, output):
    print 'Going to render', output

class MyController(TGController):
    @expose()
    @before_render(before_render_cb)
    def index(self, *args, **kw):
        return dict(page='index')
```

Or register them explicitly (useful when registering hooks on third party controllers):

```
tg.hooks.register('before_render', before_render_cb, controller=MyController.index)
```

See `tg.configuration.hooks.HooksNamespace.register()` for more details on registering hooks.

Apart from Hooks TurboGears also provide some *Configuration Milestones* you might want to have a look at to check whenever it is more proper to register an action for a configuration milestone or for an hook.

Available Hooks

- `startup()` - application wide only, called when the application starts

- `shutdown()` - application wide only, called when the application exits
- **`configure_new_app(app)` - new application got created by the application configurator.** This is the only call that can guarantee to receive the TGApp instance before any middleware wrapping.
- **`before_config(app)` -> `app` - application wide only, called right after creating application,** but before setting up most of the options and middleware. Must return the application itself. Can be used to wrap the application into middlewares that have to be executed having the full TG stack available.
- **`after_config(app)` -> `app` - application wide only, called after finishing setting everything up.** Must return the application itself. Can be used to wrap the application into middleware that have to be executed before the TG ones. Can also be used to modify the Application by mounting additional subcontrollers inside the RootController.
- `before_validate(remainder, params)` - Called before performing validation
- `before_call(remainder, params)` - Called after validation, before calling the actual controller method
- `before_render(remainder, params, output)` - Called before rendering a controller template, output is the controller return value
- `after_render(response)` - Called after finishing rendering a controller template

Notifying Custom Hooks

Custom hooks can be notified using `tg.hooks.notify`, listeners can register for any hook name, so it as simple as notifying your own hook and documenting them in your library documentation to make possible for other developers to listen for them:

```
tg.hooks.notify('custom_global_hook')
```

See `tg.configuration.hooks.HooksNamespace.notify()` for more details.

3.9.2 Controller Wrappers

Controller wrappers behave much like decorators, they sit between the controller code and TurboGears. Whenever turbogears has to call that controller it will process all the registered controller wrappers which are able to forward the request to the next in chain or just directly return an alternative value from the controller.

Registering a controller wrapper can be done using `AppConfig.register_controller_wrapper`. It is possible to register a controller wrapper for a specific controller or for the whole application, when registered to the whole application they will be applied to every controller of the application or third party libraries:

```
def controller_wrapper(next_caller):
    def call(*args, **kw):
        try:
            print 'Before handler!'
            return next_caller(*args, **kw)
        finally:
            print 'After Handler!'
    return call
```

```
base_config.register_controller_wrapper(controller_wrapper)
```

Due to the registration performance cost, controller wrappers *can only be registered before the application started*.

See `AppConfig.register_controller_wrapper()` for more details.

3.9.3 Application Wrappers

Application wrappers are like WSGI middlewares but are executed in the context of TurboGears and work with abstractions like Request and Response objects.

Application wrappers are callables built by passing the next handler in chain and the current TurboGears configuration.

They are usually subclasses of `ApplicationWrapper` which provides the expected interface.

Every wrapper, when called, is expected to accept the WSGI environment and a TurboGears context as parameters and are expected to return a `tg.request_local.Response` instance:

```
from tg.appwrappers.base import ApplicationWrapper

class AppWrapper(ApplicationWrapper):
    def __init__(self, handler, config):
        super(AppWrapper, self).__init__(handler, config)

    def __call__(self, controller, environ, context):
        print 'Going to run %s' % context.request.path
        return self.next_handler(controller, environ, context)
```

Application wrappers can be registered from you application configuration object in `app_cfg.py`:

```
base_config.register_wrapper(AppWrapper)
```

When registering a wrapper, it is also possible to specify after which other wrapper it has to run if available:

```
base_config.register_wrapper(AppWrapper, after=OtherWrapper)
```

Wrappers registered with `after=False` will run before any other available wrapper (in order of registration):

```
base_config.register_wrapper(AppWrapper, after=False)
```

See `AppConfig.register_wrapper()` for more details.

TurboGears2 CookBook

The CookBook is a collection of documentation and patterns common to TurboGears2

4.1 Upgrading TurboGears

4.1.1 Upgrading Your TurboGears Project

From 2.3.4 to 2.3.5

Genshi Work-Around available for Python3.4

Genshi 0.7 suffers from a bug that prevents it from working on Python 3.4 and causes an Abstract Syntax Tree error, to work-around this issue TurboGears provides the `templating.genshi.name_constant_patch` option that can be set to `True` to patch Genshi to work on Python 3.4.

Configuration Flow Refactoring

In previous versions the `AppConfig` object won over the *.ini file* options for practically everything, now the configurator has been modified so that `AppConfig` options are used as a template and for most options the *.ini file* wins over them.

There are still some options that are immutable and can only be defined in the `AppConfig` itself, but most of them can now be changed from the ini files.

Now the `tg.config` **object will always be reconfigured from scratch** when an application is created. Previously each time an application was created it incrementally modified the same config object leading to odd behaviours. This means that if you want a value to be available to all instances of your application you should store it in `base_config` and not in `tg.config`. This should not impact your app unless you called `AppConfig.setup_tg_wsgi_app` multiple times (which is true for test suites).

Another minor change is that `AppConfig.after_init_config` is now expected to accept a parameter with the configuration dictionary. So if you implemented a custom `after_init_config` method it is required to accept the config dictionary and make configuration changes in it.

tg.hooks is not bound to config anymore

Hooks are not bound to config anymore, but are now managed by an `HooksNamespace`. This means that they are now registered per *process and namespace* instead of being registered per-config. This leads to the same behaviour

when only one TGApp is configured per process but has a much more reliable behaviour when multiple TGApp are configured.

For most users this shouldn't cause any difference, but hooks will now be registered independently from the `tg.config` status.

Application Wrappers now provide a clearly defined interface

`ApplicationWrapper` abstract base class has been defined to provide a clear interface for application wrappers, all TurboGears provided application wrappers now adhere this interface.

I18N Translations now provided through an Application Wrapper

`I18NApplicationWrapper` now provides support for translation detection from browser language and user session. This was previously builtin into the TurboGears Dispatcher even though it was not related to dispatching itself.

The behaviour should remain the same apart from the fact that it is now executed before entering the TurboGears application and that some options got renamed:

- `lang` option has been renamed to `i18n.lang`.
- `i18n_enabled` has been renamed to `i18n.enabled`
- `beaker.session.tg_avoid_touch` option has been renamed to `i18n.no_session_touch` as it is only related to `i18n`.
- `lang_session_key` got renamed to `i18n.lang_session_key`.

For a full list of option available please refer to `I18NApplicationWrapper` itself.

Session and Cache Middlewares replaced by Application Wrappers

The `SessionMiddleware` and `CacheMiddleware` were specialized Beaker middleware for session and caching. To guarantee better integration with TurboGears and easier configuration they have been switched to Application Wrappers.

The `use_sessions=True` option got replaced by `session.enabled=True` and an additional `cache.enabled=True` option has been added.

For a full list of options refer to the `CacheApplicationWrapper` and `SessionApplicationWrapper` references.

To deactivate the application wrappers and switch back to the old middlewares, use:

```
base_config['session.enabled'] = False
base_config['use_session_middleware'] = True
```

and:

```
base_config['cache.enabled'] = False
base_config['use_cache_middleware'] = True
```

StatusRedirect middleware replaced by ErrorPageApplicationWrapper

The `StatusRedirect` middleware was inherited from Paste project, and was in charge of intercepting status codes and redirect to an error page in case of one of those.

So the `status_code_redirect=True` option got replaced by the `errorpage.enabled=True` option. For a full list of options refer to the [ErrorPageApplicationWrapper](#) reference.

As `StatusCodeRedirect` worked at WSGI level it was pretty slow and required to read the whole answer just to get the status code. Also the TurboGears context (request, response, app_globals and so on) were lost during the execution of the `ErrorController`.

In 2.3.5 this got replaced by the [ErrorPageApplicationWrapper](#), which provides the same feature using an *Application Wrappers*.

If you are still relying on `pylons.original_response` key in your `ErrorController` make sure to upgrade to the `tg.original_response` key, otherwise it won't work anymore.

The change should be transparent for most users, in case you want to get back the old `StatusCodeRedirect` behaviour you use the following option:

```
base_config['status_code_redirect'] = True
```

Keep in mind that the other options from [ErrorPageApplicationWrapper](#) apply and are converted to options for the `StatusCodeRedirect` middleware.

Transaction Manager is now an application wrapper

Transaction Manager (the component in charge of committing or rolling back your sqlalchemy transaction) is now replaced by [TransactionApplicationWrapper](#) which is an application wrapper in charge of committing or rolling back the transaction.

So the `use_transaction_manager=True` option got replaced by the `tm.enabled=True` option. For a full list of options refer to the [TransactionApplicationWrapper](#) reference.

There should be no behavioural changes with this change, the only difference is now that the transaction manager applies before the WSGI middlewares as it is managed by TurboGears itself. So if your application was successful and there was an error in a middleware that happens after (for example `ToscaWidgets` resource injection) the transaction will be committed anyway as the code that created the objects and for which they should be committed was successful.

If you want to recover back the *old TGTransactionManager middleware* you can use the following option:

```
base_config['use_transaction_manager'] = True
```

TurboGears provides its own ming ODMSession manager as an Application Wrapper

The major change is that [MingApplicationWrapper](#) now behaves like SQLAlchemy session when streaming responses.

The session is automatically flushed for you at the end of the request, in case of stramed responses instead you will have to manually manage the session yourself if it is used inside the response generator as specified in [Streaming Response](#).

To recover the previous behavior set `ming.autoflush=False` and replace the `AppConfig.add_ming_middleware` method with the following:

```
def add_ming_middleware(self, app):
    import ming.odm.middleware
    return ming.odm.middleware.MingMiddleware(app)
```

From 2.3.3 to 2.3.4

JSON Support no longer supports simplegeneric

To provide support for customization the `json.isodates` and `json.custom_encoders` options are now available during application configuration. Those are also available in `@expose('json') render_params`, see *JSON and JSONP Rendering*.

lang option is now fallback when i18n is enabled

TurboGears provided a `lang` configuration option which was only meaningful when `i18n` was disabled with `i18n_enabled = False`. The `lang` option would force the specified language for the whole web app, independently from user session or browser languages.

Now the `lang` option when specified is used as the fallback language when `i18n` is actually enabled (which is the default).

tg.util is now officially public

As `tg.util` provided utilities that could be useful to app developers the module has been cleaned up keeping only public features and is now documented at `tg.util`

From 2.3.2 to 2.3.3

abort can now skip error/document and authentication

`tg.controllers.util.abort()` can now provide a pass-through abort which will answer as is instead of being intercepted by authentication layer to redirect to login page or by Error controller to show a custom error page. This can be helpful when writing API responses that should just provide output as is.

@require can now be used for allow_only

It is now possible to use `tg.decorators.require()` as value for controllers `allow_only` to enable `smart_denial` or provide a custom `denial_handler` for *Controller-level authorization*

@require is now a TurboGears decoration

`@require` decorator is now a TurboGears decoration, the order it is applied won't matter anymore if other decorators are placed on the controller.

@beaker_cache is now replaced by @cached

`@beaker_cache` decorator was meant to work on plain function, the new `@cached` decorator is meant to work explicitly on TurboGears controllers. The order the decorator is applied won't matter anymore just like the other turbogears decorations.

`@beaker_cache` is still provided, but it's use on controllers is discouraged.

controller_wrappers now get config on call and not on construction

Whenever a controller wrapper is registered it won't get the `app_config` parameter anymore on construction, instead it will receive the configuration as a parameter each time it is called.

The controller wrapper signature has changed as following:

```
def controller_wrapper(next_caller):
    def call(config, controller, remainder, params):
        return next_caller(config, controller, remainder, params)
    return call
```

If you still need to access the application configuration into the controller wrapper constructor, use `tg.config`.

TurboGears will try to setup the controller wrapper with the new method signature, if it fails it will fallback to the old controller wrappers signature and provide a *DeprecationWarning*.

get_lang always returns a list

Since 2.3.2 `get_lang` supports the `all` option, which made possible to ask TurboGears for all the languages requested by the user to return only those for which the application supports translation (`all=False`).

When `get_lang(all=True)` was called, two different behaviors where possible: Usually the whole list of languages requested by the user was returned, unless the application supported no translations. In that case `None` was returned.

Now `get_lang(all=True)` behaves in a more predictable way and always returns the whole list of languages requested by the user. In case `i18n` is not enabled an empty list is returned.

From 2.3.1 to 2.3.2

Projects quickstarted on 2.3 should work out of the box.

Kajiki support for TW2 removed

If your application is using Kajiki as its primary rendering engine, TW2 widget will now pick the first supported engine instead of Kajiki.

This is due to the fact that recent TW2 version removed support for Kajiki.

AppConfig.setup_mimetypes removed

If you were providing custom mimetypes by overriding the `setup_mimetypes` method in `AppConfig` this is not supported anymore. To register custom mimetypes just declare them in `base_config.mimetype_lookup` dictionary in your `config/app_cfg.py`.

Custom rendering engines support refactoring

If you were providing a custom rendering engine through `AppConfig.setup_NAME_renderer` methods, those are now deprecated. While they should continue to work it is preferred to update your rendering engine to the new factory based `tg.configuration.AppConfig.register_rendering_engine()`

Chameleon Genshi support is now provided by an extension

Chameleon Genshi rendering support is now provided by `tgext.chameleon_genshi` instead of being built-in inside TurboGears itself.

Validation error_handlers now call their hooks and wrappers

Previous to 2.3.2 controller methods when used as `error_handlers` didn't call their registered hooks and controller wrappers, not if an hook or controller wrapper is attached to an error handler it will correctly be called. Only exception is `before_validate` hook as `error_handlers` are not validated.

AppConfig.add_dbsession_removal_middleware renamed

If you were providing a custom `add_dbsession_removal_middleware` method you should now rename it to `add_sqlalchemy_middleware`.

Error Reporting options grouped in .ini file

Error reporting options have been grouped in `trace_errors` options.

While previous option names continue to work for backward compatibility, they will be removed in future versions. Email error sending options became:

```
trace_errors.error_email = you@yourdomain.com
trace_errors.from_address = turbogears@localhost
trace_errors.smtp_server = localhost
```

```
trace_errors.smtp_use_tls = true
trace_errors.smtp_username = unknown
trace_errors.smtp_password = unknown
```

From 2.3 to 2.3.1

Projects quickstarted on 2.3 should work out of the box.

AppConfig.register_hook Deprecation

`register_hook` function in application configuration got deprecated and replaced by `tg.hooks.register` and `tg.hooks.wrap_controller`.

`register_hook` will continue to work like before, but will be removed in future versions. Check [Hooks Guide](#) and upgrade to `tg.hooks` based hooks to avoid issues on `register_hook` removal.

Exposition and Wrappers now resolved lazily

Due to [Configuration Milestones](#) support controller exposition is now resolved lazily when the configuration process has setup the renderers. This enables a smarter exposition able to correctly behave even when controllers are declared before the application configuration.

Application wrappers dependencies are now solved lazily too, this makes possible to reorder them before applying the actual wrappers so that the order of registration doesn't matter when a wrapper ordering is specified.

Some methods in AppConfig got renamed

To provide a cleaner distinction between methods users are expected to subclass to customize the configuration process and methods which are part of TurboGears setup itself.

Validation error reporting cleanup

TurboGears always provided information on failed validations in a unorganized manner inside `tmpl_context.form_errors` and other locations.

Validation information are now reported in `request.validation` dictionary all together. `tmpl_context.form_errors` and `tmpl_context.form_values` are still available but deprecated.

From 2.2 to 2.3

Projects quickstarted on 2.2 should mostly work out of the box.

GearBox replaced PasteScript

Just by installing `gearbox` itself your TurboGears project will be able to use `gearbox` system wide commands like `gearbox serve`, `gearbox setup-app` and `gearbox makepackage` commands. These commands provide a replacement for the `paster serve`, `paster setup-app` and `paster create` commands.

The main difference with the `paster` command is usually only that `gearbox` commands explicitly set the configuration file using the `--config` option instead of accepting it positionally. By default `gearbox` will always load a configuration file named `development.ini`, this mean you can simply run `gearbox serve` in place of `paster serve development.ini`

Gearbox HTTP Servers If you are moving your TurboGears2 project from `paster` you will probably end serving your application with Paste HTTP server even if you are using the `gearbox serve` command.

The reason for this behavior is that `gearbox` is going to use what is specified inside the `server:main` section of your `.ini` file to serve your application. TurboGears2 projects quickstarted before 2.3 used Paste and so the projects is probably configured to use `Paste#http` as the server. This is not an issue by itself, it will just require you to have Paste installed to be able to serve the application, to totally remove the Paste dependency simply replace **Paste#http** with **gearbox#wsgiref**.

Enabling GearBox migrate and tgshell commands To enable `gearbox migrate` and `gearbox tgshell` commands make sure that your `setup.py entry_points` look like:

```
entry_points={
    'paste.app_factory': [
        'main = makonoauth.config.middleware:make_app'
    ],
    'gearbox.plugins': [
        'turbogears-devtools = tg.devtools'
    ]
}
```

The `paste.app_factory` section will let `gearbox serve` know how to create the application that has to be served. Gearbox relies on PasteDeploy for application setup, so it required a `paste.app_factory` section to be able to correctly load the application.

While the **gearbox.plugins** section will let *gearbox* itself know that inside that directory the `tg.devtools` commands have to be enabled making `gearbox tgshell` and `gearbox migrate` available when we run `gearbox` from inside our project directory.

Removing Paste dependency When performing `python setup.py develop` you will notice that Paste will be installed. To remove such dependency you should remove the `setup_requires` and `paster_plugins` entries from your `setup.py`:

```
setup_requires=["PasteScript >= 1.7"],
paster_plugins=['PasteScript', 'Pylons', 'TurboGears2', 'tg.devtools']
```

WebHelpers Dependency

If your project used WebHelpers, the package is not a turbogears dependency anymore, you should remember to add it to your `setup.py` dependencies.

Migrations moved from sqlalchemy-migrate to Alembic

Due to sqlalchemy-migrate not supporting SQLAlchemy 0.8 and Python 3, the migrations for newly quickstarted projects will now rely on Alembic. The migrations are now handled using `gearbox migrate` command, which supports the same subcommands as the `paster migrate` one.

The `gearbox sqla-migrate` command is also provided for backward compatibility for projects that need to keep using sqlalchemy-migrate.

Pagination module moved from tg.paginate to tg.support.paginate

The pagination code, which was previously imported from webhelpers, is now embedded in the TurboGears distribution, but it changed its exact location. If you are using `tg.paginate.Page` manually at the moment, you will have to fix your imports to be `tg.support.paginate.Page`.

Anyway, you should preferably use the decorator approach with `tg.decorators.paginate` - then your code will be independent of the TurboGears internals.

From 2.1 to 2.2

Projects quickstarted on 2.1 should mostly work out of the box.

Main points of interest when upgrading from 2.1 to 2.2 are related to some features deprecated in 2.1 that now got removed, to the new ToscaWidgets2 support and to the New Authentication layer.

Both ToscaWidgets2 and the new auth layer are disabled by default, so they should not get in your way unless you explicitly want.

Deprecations now removed

`tg.url` changed in release 2.1, in 2.0 parameters for the url could be passed as parameters for the `tg.url` function. This continued to work in 2.1 but provided a `DeprecationWarning`. Since 2.1 parameters to the url call must be passed in the `params` argument as a dictionary. Support for url parameters passed as arguments have been totally removed in 2.2

`use_legacy_renderer` option isn't supported anymore. Legacy renderers (Buffets) got deprecated in previous versions and are not available anymore in 2.2.

`__before__` and `__after__` controller methods got deprecated in 2.1 and are not called anymore, make sure you switched to the new `_before` and `_after` methods.

Avoiding ToscaWidgets2

If you want to keep using ToscaWidgets1 simply don't install ToscaWidgets2 in your environment.

If your project has been quickstarted before 2.2 and uses ToscaWidgets1 it can continue to work that way, by default projects that don't enable tw2 in any way will continue to use ToscaWidgets1.

If you install tw2 packages in your environment the admin interface, sprox, crud and all the functions related to form generation will switch to ToscaWidgets2. This will force you to enable tw2 with the `use_toscawidgets2` option, otherwise they will stop working.

So if need to keep using ToscaWidgets1 only, don't install any tw2 package.

Mixing ToscaWidgets2 and ToscaWidgets1

Mixing the two widgets library is perfectly possible and can be achieved using both the `use_toscawidgets` and `use_toscawidgets2` options. When ToscaWidgets2 is installed the admin, sprox and the crud controller will switch to tw2, this will require you to enable the `use_toscawidgets2` option.

If you manually specified any widget inside Sprox forms or CrudRestController you will have to migrate those to tw2. All the other forms in your application can keep being ToscaWidgets1 forms and widgets.

Moving to ToscaWidgets2

Switching to tw2 can be achieved by simply placing the `prefer_toscawidgets2` option in your `config/app_cfg.py`. This will totally disable ToscaWidgets1, being it installed or not. So all your forms will have to be migrated to ToscaWidgets2.

New Authentication Layer

2.2 release introduced a new authentication layer to support repoze.who v2 and prepare for moving forward to Python3. When the new authentication layer is not in use, the old one based on repoze.what, repoze.who v1 and repoze.who-testutil will be used.

As 2.1 applications didn't explicitly enable the new authentication layer they should continue to work as before.

Switching to the new Authentication Layer

Switching to the new authentication layer should be quite straightforward for applications that didn't customize authentication. The new layer gets enabled only when a `base_config.sa_auth.authmetadata` object is present inside your `config/app_cfg.py`.

To switch a plain project to the new authentication layer simply add those lines to your `app_cfg.py`:

```
from tg.configuration.auth import TGAAuthMetadata

#This tells to TurboGears how to retrieve the data for your user
class ApplicationAuthMetadata(TGAAuthMetadata):
    def __init__(self, sa_auth):
        self.sa_auth = sa_auth
    def get_user(self, identity, userid):
        return self.sa_auth.db.session.query(self.sa_auth.user_class).filter_by(user_name=userid).first()
    def get_groups(self, identity, userid):
        return [g.group_name for g in identity['user'].groups]
    def get_permissions(self, identity, userid):
        return [p.permission_name for p in identity['user'].permissions]

base_config.sa_auth.authmetadata = ApplicationAuthMetadata(base_config.sa_auth)
```

If you customized authentication in any way, you will probably have to port forward all your customizations, in this case, if things get too complex you can keep remaining on the old authentication layer, things will continue to work as before.

After enabling the new authentication layer you will have to switch your repoze.what imports to tg imports:

```
#from repoze.what import predicates becomes
from tg import predicates
```

All the predicates previously available in repoze.what should continue to be available. Your project should now be able to upgrade to repoze.who v2, before doing that remember to remove the following packages which are not in use anymore and might conflict with repoze.who v2:

- repoze.what
- repoze.what.plugins.sql
- repoze.what-pylons
- repoze.what-quickstart
- repoze.who-testutil

The only repoze.who packages you should end up having installed are:

- repoze.who-2.0
- repoze.who.plugins.sa
- repoze.who_friendlyform

4.2 Basic Recipes

4.2.1 Creating and Validating Forms

TurboGears relies on ToscaWidgets for Forms building and validations. Since version 2.2 TurboGears uses ToscaWidgets2, this is an introduction on using ToscaWidgets2 for building and validating forms, a more complete documentation is available on the [ToscaWidgets2 Documentation](#) itself.

Displaying Forms

To create a form you will have to declare it specifying:

- the form action (where to submit the form data)
- the form layout (how the form will be displayed)
- the form fields

The *action* can be specified as an attribute of the form itself, while the *layout* must be a class named **child** which has to inherit from `tw2.forms.BaseLayout`. Any of `tw2.forms.TableLayout` or `tw2.forms.ListLayout` will usually do, but you can easily write your own custom layouts. The form *fields* can then be specified inside the **child** class.

```
import tw2.core as twc
import tw2.forms as twf

class MovieForm(twf.Form):
    class child(twf.TableLayout):
        title = twf.TextField()
        director = twf.TextField(value='Default Director')
        genres = twf.CheckBoxList(options=['Action', 'Comedy', 'Romance', 'Sci-fi'])

    action = '/save_movie'
```

To display the form we can return it from the controller where it must be rendered:

```
@expose('tw2test.templates.index')
def index(self, *args, **kw):
    return dict(page='index', form=MovieForm)
```

and *display* it inside the template itself. Any field of the form can be filled using the *value* argument passed to the display function. The values provided inside this argument will override the field default ones.

```
<div id="getting_started">
    ${form.display(value=dict(title='default title'))}
</div>
```

When submitting the form the **save_movie** controller declared in the *action* attribute of the form will receive the submitted values as any other provided GET or POST parameter.

```
@expose()
def save_movie(self, **kw):
    return str(kw)
```

Validating Fields

ToscaWidgets2 is able to use any *FormEncode* validator for validation of both fields and forms. More validators are also provided inside the `tw2.core.validators` module.

To start using validation we have to declare the validator for each form field. For example to block submission of our previous form when no title or director is provided we can use the `tw2.core.Required` validator:

```
class MovieForm(twf.Form):
    class child(twf.TableLayout):
        title = twf.TextField(validator=twc.Required)
        director = twf.TextField(value="Default Director", validator=twc.Required)
        genres = twf.CheckBoxList(options=['Action', 'Comedy', 'Romance', 'Sci-fi'])

    action = '/save_movie'
```

Now the forms knows how to validate the title and director fields, but those are not validated in any way. To enable validation in TurboGears we must use the **tg.validate** decorator and place it at our form action:

```
@expose()
@validate(MovieForm, error_handler=index)
def save_movie(self, *args, **kw):
    return str(kw)
```

Now every submission to `/save_movie` url will be validated against the `MovieForm` and if it doesn't pass validation will be redirected to the `index` method where the form will display an error for each field not passing validation.

More about TurboGears support for validation is available inside the [TurboGears Validation](#) page.

Validating Compound Fields

Suppose that you are afraid that people might enter a wrong director name for your movies. The most simple solution would be to require them to enter the name two times to be sure that it is actually the correct one.

How can we enforce people to enter two times the same name inside our form? Apart from fields, `ToscaWidgets` permits to set validators to forms. Those can be used to validate form fields together instead of one by one. To check that our two directors equals we will use the `formencode.validators.FieldsMatch` validator:

```
import tw2.core as twc
import tw2.forms as twf
from formencode.validators import FieldsMatch

class MovieForm(twf.Form):
    class child(twf.TableLayout):
        title = twf.TextField(validator=twc.Required)
        director = twf.TextField(value="Default Director", validator=twc.Required)
        director_verify = twf.TextField()
        genres = twf.CheckBoxList(options=['Action', 'Comedy', 'Romance', 'Sci-fi'])

    action = '/save_movie'
    validator = FieldsMatch('director', 'director_verify')
```

Nothing else of our code needs to be changed, our `/save_movie` controller already has validation for the `MovieForm` and when the form is submitted after checking that there is a title and director will also check that both `director` and `director_verify` fields equals.

Relocatable Widget Actions

Whenever you run your application on a mount point which is not the root of the domain name your actions will have to point to the right path inside the mount point.

In TurboGears2 this is usually achieved using the `tg.url` function which checks the `SCRIPT_NAME` inside the request environment to see where the application is mounted. The issue with widget actions is that widget actions are globally declared and `tg.url` cannot be called outside of a request.

Calling `tg.url` while declaring a form and its action will cause a crash to avoid this TurboGears provides a lazy version of the `url` method which is evaluated only when the widget is displayed (`tg.lurl`):

```
from tg import lurl

class MovieForm(twf.Form):
    class child(twf.TableLayout):
        title = twf.TextField(validator=twc.Required)
        director = twf.TextField(value="Default Director", validator=twc.Required)
        genres = twf.CheckBoxList(options=['Action', 'Comedy', 'Romance', 'Sci-fi'])
```

```
action = lurl('/save_movie')
```

Using `tg.lurl` the form action will be correctly written depending on where the application is mounted.

Please pay attention that usually when registering resources on `ToscaWidgets` (both `tw1` and `tw2`) it won't be necessary to call neither `tg.url` or `tg.lurl` as all the `Link` subclasses like `JSLink`, `CSSLink` and so on will already serve the resource using the application mount point.

Custom Layouts

While using `tw2.forms.TableLayout` and `tw2.forms.ListLayout` it's easy to perform most simple styling and customization of your forms, for more complex widgets a custom template is usually the way to go.

You can easily provide your custom layout by subclassing `tw2.forms.widgets.BaseLayout` and declaring a template for it inside your forms.

For example it is possible to create a name/surname form with a side field for notes using the bootstrap CSS framework:

```
from tw2.core import Validator
from tw2.forms.widgets import Form, BaseLayout, TextField, TextArea, SubmitButton

class SubscribeForm(Form):
    action = '/submit'

    class child(BaseLayout):
        inline_engine_name = 'genshi'
        template = '''
<div xmlns:py="http://genshi.edgewall.org/"
    py:strip="">
    <py:for each="c in w.children_hidden">
        ${c.display()}
    </py:for>

    <div class="form form-horizontal">
        <div class="form-group">
            <div class="col-md-7">
                <div py:with="c=w.children.name"
                    class="form-group ${c.error_msg and 'has-error' or ''}">
                    <label for="${c.compound_id}" class="col-md-3 control-label">${c.label}</label>
                    <div class="col-md-9">
                        ${c.display()}
                        <span class="help-block" py:content="c.error_msg"/>
                    </div>
                </div>
            </div>
            <div py:with="c=w.children.surname"
                class="form-group ${c.error_msg and 'has-error' or ''}">
                <label for="${c.compound_id}" class="col-md-3 control-label">${c.label}</label>
                <div class="col-md-9">
                    ${c.display()}
                    <span class="help-block" py:content="c.error_msg"/>
                </div>
            </div>
        </div>
        <div class="col-md-4 col-md-offset-1">
            ${w.children.notes.display()}
        </div>
    </div>
</div>'''
```

```
</div>
'''

name = TextField(label=l_('Name'), validator=Validator(required=True),
                 css_class="form-control")
surname = TextField(label=l_('Surname'), validator=Validator(required=True),
                   css_class="form-control")
notes = TextArea(label=None, placeholder=l_("Notes"),
                 css_class="form-control", rows=8)

submit = SubmitButton(css_class='btn btn-primary', value=l_('Create'))
```

4.2.2 JSON and JSONP Rendering

JSON Renderer

TurboGears always provided builtin support for JSON Rendering, this is provided by the `JSONRenderer` and the `json.encode()` function.

The first is what empowers the `@expose('json')` feature while the second is an utility function you can call whenever encoding to json is needed. Both rely on `tg.jsonify.JSONEncoder` which is able to handle more types than the standard one provided by the python `json` module and can be extended to support more types.

Using it is as simple as:

```
@expose('json')
def jp(self, **kwargs):
    return dict(hello='World')
```

Which, when calling `/jp` would result in:

```
{"hello": "World"}
```

Customizing JSON Encoder

While you can create your own encoder, turbogears has a default instance of `JSONEncoder` which is used for all encoding performed by the framework itself. Behavior of this encoder can be driven by providing a `__json__` method inside objects for which you want to customize encoding and can be configured using `AppConfig` which supports the following options:

- `json.isodates` -> Whenever to encode dates in ISO8601 or not, the default is `False`
- `json.custom_encoders` -> Dictionary of type: `function mappings` which can specify custom encoders for specific types. Custom encoders are functions that are called to get a basic object the json encoder knows how to handle.

For example to configure a custom encoder for dates your project `app_cfg.py` would look like:

```
from datetime import date

def dmy_encoded_date(d):
    return d.strftime('%d/%m/%Y')

base_config['json.custom_encoders'] = {date: dmy_encoded_date}
```

That would cause all `datetime.date` instances to be encoded using `dmy_encode_date` function.

If the encoded object provides a `__json__` method this is considered the **custom encoder** for the object itself and it is called to get a basic type the json encoder knows how to handle (usually a `dict`).

Note: `json.custom_encoders` take precedence over `__json__`, this is made so that users can override behavior for third party objects that already provide a `__json__` method.

Per method customization

The same options available inside the `json.` configuration namespace are available as `render_params` for the `expose` decorator. So if you want to turn on/off iso formatted dates for a single method you can do that using:

```
from datetime import datetime

@expose('json', render_params=dict(isodates=True))
def now(self, **kwargs):
    return dict(now=datetime.utcnow())
```

JSONP Renderer

Since version 2.3.2 TurboGears provides built-in support for JSONP rendering.

JSONP works much like JSON output, but instead of providing JSON response it provides an application/javascript response with a call to a javascript function providing all the values returned by the controller as function arguments.

To enable JSONP rendering you must first append it to the list of required engines inside your application `config/app_cfg.py`:

```
base_config.renderers.append('jsonp')
```

Then you can declare a JSONP controller by exposing it as:

```
@expose('jsonp')
def jp(self, **kwargs):
    return dict(hello='World')
```

When accessing `/jp?callback=callme` you should see:

```
callme({"hello": "World"});
```

If you omit the `callback` parameter an error will be returned as it is required to know the callback name when using JSONP.

Custom callback parameter

By default TurboGears will expect the callback name to be provided in a `callback` parameter. This parameter has to be accepted by your controller (otherwise you can use `**kwargs` like the previous examples).

If you need to use a different name for the callback parameter just provide it in the `render_params` of your exposition:

```
@expose('jsonp', render_params={'callback_param': 'call'})
def jp(self, **kwargs):
    return dict(hello='World')
```

Then instead of opening `/jp?callback=callme` to get the JSONP response you will need to open `/jp?call=callme` as stated by the `callback_param` option provided in the `render_params`.

Exposing both JSON and JSONP

If you want to expose a controller as both JSON and JSONP, just provide both expositions. You can then use TurboGears request extensions support to choose which response you need:

```
@expose('json')
@expose('jsonp')
def jp(self, **kwargs):
    return dict(hello='World')
```

To get the JSON response simply open `/jp.json` while to get the JSONP response go to `/jp.js?callback=callme`. If no extension is provided the first exposition will be returned (in this case JSON).

4.2.3 DataGrid Tutorial

DataGrid is a quick way to present data in tabular form.

The columns to put inside the table are specified with the `fields` constructor argument in a list. Each entry of the list can be an accessor (attribute name or function), a tuple (title, accessor) or a `tw2.forms.datagrid.Column` instance.

Preparing Application

This tutorial will show an addressbook with a set of people each with a name, surname and phone number. Model used will be:

```
from sqlalchemy import Table, ForeignKey, Column
from sqlalchemy.types import Unicode, Integer, DateTime

class Person(DeclarativeBase):
    __tablename__ = 'person'

    uid = Column(Integer, autoincrement=True, primary_key=True)
    name = Column(Unicode(255))
    surname = Column(Unicode(255))
    phone = Column(Unicode(64))
```

and we will populate with this data:

```
for i in [['John', 'Doe', '3413122314'],
         ['Lucas', 'Darkstone', '378321322'],
         ['Dorian', 'Gray', '31337433'],
         ['Whatever', 'Person', '3294432321'],
         ['Aaliyah', 'Byron', '676763432'],
         ['Caesar', 'Ezra', '9943243243'],
         ['Fahd', 'Gwyneth', '322313232'],
         ['Last', 'Guy', '23132321']]:
    DBSession.add(Person(name=i[0], surname=i[1], phone=i[2]))
```

Basic DataGrid

With a model and some data set up, we can now start declaring our DataGrid and the fields it has to show:

```
from tw2.forms import DataGrid

addressbook_grid = DataGrid(fields=[
    ('Name', 'name'),
    ('Surname', 'surname'),
    ('Phone', 'phone')
])
```

After declaring the grid itself we will need to fetch the data to show inside the grid from our controller. For this example we will do it inside the `RootController.index` method:

```
@expose('dgridt.templates.index')
def index(self):
    data = DBSession.query(Person)
    return dict(page='index', grid=addressbook_grid, data=data)
```

Now the grid can be displayed in the template like this:

Template code necessary to show the grid in `templates/index.html`:

```
<div>${grid.display(value=data)}</div>
```

Paginating DataGrid

Now that the grid can be displayed next probable improvement would be to paginate it. Displaying 10 results is fine, but when results start to grow it might cause performance problems and make results harder to view.

The same things explained in the *Pagination in TurboGears* tutorial apply here. First of all it is needed to adapt the controller method to support pagination:

```
from tg.decorators import paginate

@expose('dgridt.templates.index')
@paginate("data", items_per_page=3)
def index(self):
    data = DBSession.query(Person)
    return dict(page='index', grid=addressbook_grid, data=data)
```

If you run the application now you will see only 3 results as they get paginated three by three and we are still missing a way to change page. What is needed now is a way to switch pages and this can be easily done as the `paginate` decorator adds to the template context a `paginator` variable where all the paginators currently available are gathered. Rendering the “data” paginator somewhere inside the template is simply enough to have a working pagination for our datagrid.

Template in `templates/index.html` would become:

```
<div>${grid.display(value=data)}</div>
<div>${tpl_context.paginator.data.paginator()}</div>
```

Now the page should render with both the datagrid and the pages under the grid itself, making possible to switch between the pages.

Sorting Columns

DataGrid itself does not provide a way to implement columns sorting, but it can be easily achieved by inheriting from `tw2.forms.datagrid.Column` to add a link that can provide sorting.

First of all we need to declare or `SortableColumn` class that will return the link with the sorting request as the title for our DataGrid:

```
from sqlalchemy import asc, desc
from tw2.forms.datagrid import Column
import genshi

class SortableColumn(Column):
    def __init__(self, title, name):
        super(SortableColumn, self).__init__(name)
        self._title_ = title

    def set_title(self, title):
        self._title_ = title

    def get_title(self):
        current_ordering = request.GET.get('ordercol')
        if current_ordering and current_ordering[1:] == self.name:
            current_ordering = '-' if current_ordering[0] == '+' else '+'
        else:
            current_ordering = '+'
        current_ordering += self.name

        new_params = dict(request.GET)
        new_params['ordercol'] = current_ordering

        new_url = url(request.path_url, params=new_params)
        return genshi.Markup('<a href="%s">%s</a>' % dict(page_url=new_url, title=self._title_))

    title = property(get_title, set_title)
```

It is also needed to tell to the DataGrid that it has to use the `SortableColumn` for its fields:

```
addressbook_grid = DataGrid(fields=[
    SortableColumn('Name', 'name'),
    SortableColumn('Surname', 'surname'),
    SortableColumn('Phone', 'phone')
])
```

Now if we reload the page we should see the clickable links inside the headers of the table, but if we click one the application will crash because of an unexpected argument. We are now passing the `ordercol` argument to our constructor to tell it for which column we want the data to be ordered and with which ordering.

To handle the new parameter the controller must be modified to accept it and perform the ordering:

```
@expose('dgridt.templates.index')
@paginate("data", items_per_page=3)
def index(self, *args, **kw):
    data = DBSession.query(Person)
    ordering = kw.get('ordercol')
    if ordering and ordering[0] == '+':
        data = data.order_by(asc(ordering[1:]))
    elif ordering and ordering[0] == '-':
        data = data.order_by(desc(ordering[1:]))
    return dict(page='index', grid=addressbook_grid, data=data)
```

Now the ordering should work and clicking two times on a column should invert the ordering.

Edit Column Button

DataGrid also permits to pass functions in the *fields* parameter to build the row content. This makes possible for example to add an *Actions* column where to put an edit button to edit the entry on the row.

To perform this it is just required to add another field with the name and the function that will return the edit link. In this example `addressbook_grid` would become:

```
addressbook_grid = DataGrid(fields=[
    SortableColumn('Name', 'name'),
    SortableColumn('Surname', 'surname'),
    SortableColumn('Phone', 'phone'),
    ('Action', lambda obj:genshi.Markup('<a href="%s">Edit</a>' % url('/edit', params=dict(item_id=obj.id))))
])
```

4.2.4 TurboGears Automatic CRUD Generation

Overview

This is a simple extension that provides a basic controller class that can be extended to meet the needs of the developer. The intention is to provide a fast path to data management by allowing the user to define forms and override the data interaction with custom manipulations once the view logic is in place. The name of this extensible class is `CrudRestController`.

What is CRUD?

CRUD is a set of functions to manipulate the data in a database: create, read, update, delete.

Um, REST?

REST is a methodology for mapping resource manipulation to meaningful URL. For instance if we wanted to edit a user with the ID 3, the URL might look like: `/users/3/edit`. For a brief discussion on REST, take a look at the [microformats entry](#).

Before We Get Started

Here is the model definition we will be using for this tutorial:

```
from sqlalchemy import Column, Integer, String, Date, Text, ForeignKey
from sqlalchemy.orm import relation

from moviedemo.model import DeclarativeBase

class Genre(DeclarativeBase):
    __tablename__ = "genres"
    genre_id = Column(Integer, primary_key=True)
    name = Column(String(100))

class Movie(DeclarativeBase):
    __tablename__ = "movies"
```

```
movie_id = Column(Integer, primary_key=True)
title = Column(String(100), nullable=False)
description = Column(Text, nullable=True)
genre_id = Column(Integer, ForeignKey('genres.genre_id'))
genre = relation('Genre', backref='movies')
release_date = Column(Date, nullable=True)
```

EasyCrudRestController

The first thing we want to do is instantiate a `EasyCrudRestController`. We import the controller from the extension, and then provide it with a model class that it will use for its data manipulation. For this example we will utilize the `Movie` class.:

```
from tgext.crud import EasyCrudRestController
from moviedemo.model import DBSession, Movie

class MovieController(EasyCrudRestController):
    model = Movie

class RootController(BaseController):
    movies = MovieController(DBSession)
```

That will provide a simple and working CRUD controller already configured with some simple views to list, create, edit and delete objects of type `Movie`.

Customizing EasyCrudRestController

The `EasyCrudRestController` provides some quick customization tools. Having been thought to quickly prototype parts of your web applications the `EasyCrudRestController` permits both to tune forms options and to add utility methods on the fly:

```
class MovieController(EasyCrudRestController):
    model = Movie

    title = "My admin title"

    __form_options__ = {
        '__hide_fields__': ['movie_id'],
        '__field_order__': ['title', 'description'],
        '__field_widget_types__': {'description': TextArea}
    }

    __table_options__ = { # see Sprox TableBase and Sprox TableFiller
        '__limit_fields__': ['title', 'desc'],
        '__add_fields__': {'computed': None},
        'computed': lambda filler, row: row.some_field * 2
    }

    __setters__ = {
        'release': ('release_date', datetime.datetime.utcnow)
    }
```

The `title` option provides a way to customize the title displayed in the titlebar of your browser.

The `__form_options__` dictionary will permit to tune the forms configuration. The specified options will be applied to both the form used to create new entities and to edit the existing ones. To have a look at the available

options refer to [Sprox FormBase](#)

The `__table_options__` dictionary will permit to tune the forms configuration. To have a look at the available options refer to [Sprox TableBase](#), [Sprox TableFiller](#), and their parents as well.

The `__setters__` option provides a way to add new simple methods on the fly to the controller. The key of the provided dictionary is the name of the method, while the value is a tuple where the first argument is the attribute of the object that has to be changed. The second argument is the value that has to be set, if the second argument is a callable it will be called passing the object to edit as the argument.

In the previous example calling <http://localhost:8080/movies/5/release> will mark the movie 5 as released today.

Enabling SubString Searches

The `CrudRestController` provides ready to use search function, when opening the controller index you will see a list of entries and a search box.

By default the search box looks for perfect matches, this is often not the case especially if you are looking in long text entries that the user might not remember, this behavior can be changed by using the `substring_filters` option.

You can enable substring searches for all the text fields by setting it to `True`:

```
class MovieController(EasyCrudRestController):
    model = Movie
    substring_filters = True

    __table_options__ = {
        '__omit_fields__': ['movie_id'],
    }
```

This will permit to search for text inside our movies title and descriptions. If you want to restrict substring searches to only some fields you can specify them explicitly:

```
class MovieController(EasyCrudRestController):
    model = Movie
    substring_filters = ['description']

    __table_options__ = {
        '__omit_fields__': ['movie_id'],
    }
```

Remembering Previous Values

The default behavior of the `CrudRestController` is to set fields to the submitted value, if the user submits an empty value the object property gets emptied, there are cases where you might prefer it to keep the previous value when an empty one is provided. This behavior can be enabled using the `remember_values` option.

This is specially the case with images, you usually prefer to keep the previous image if a new one is not provided instead of deleting it at all.

Suppose we have a `Photo` model which has an `image` field using `tgext.datahelpers.AttachedImage` to provide an image field (please refer to [tgext.datahelpers documentation](#) for more details). By default each time the user submits the edit form without specifying a new image we would lose our previous image, to avoid this behavior and just keep our previous image when none is specified we can use the `remember_values` option:

```
class PhotoManageController(EasyCrudRestController):
    model = Photo
    remember_values = ['image']
```

```
__table_options__ = {
    '__omit_fields__': ['uid'],
    '__xml_fields__': ['image'],

    'image': lambda filler, row: Markup('' % row.image.thumb_url) if row.image else ''
}

__form_options__ = {
    '__field_widget_types__': {'image': FileField},
    '__field_validator_types__': {'image': FieldStorageUploadConverter},
    '__field_widget_args__': {'image': {'label': 'Photo PNG (640x280)'}},
    '__hide_fields__': ['uid']
}
```

Customizing Pagination

The `CrudRestController` provides pagination support, by default this is enabled and provides 7 entries per page.

To tune pagination you can set the `pagination` set of options. To change the number of entries displayed you can set `pagination['items_per_page']`.

To display 20 items per page you can for example use:

```
class MovieController(EasyCrudRestController):
    model = Movie
    pagination = {'items_per_page': 20}
```

To totally disable pagination just set the `pagination` option to `False`:

```
class MovieController(EasyCrudRestController):
    model = Movie
    pagination = False
```

Custom CrudRestController

The `EasyCrudRestController` provides a preconfigured `CrudRestController` but often you will need to deeply customize it for your needs. To do that we can start over with a clean controller and start customizing it:

```
from tgext.crud import CrudRestController
from moviedemo.model import DBSession, Movie

class MovieController(CrudRestController):
    model = Movie

class RootController(BaseController):
    movies = MovieController(DBSession)
```

Well that won't actually get you anywhere, in fact, it will do nothing at all. We need to provide `CrudRestController` with a set of widgets and datafillers so that it knows how to handle your REST requests. First, let's get all of the Movies to display in a table.

Sprox

`Sprox` is a library that can help you to generate forms and filler data. It utilizes metadata extracted from the database definitions to provide things like form fields, drop downs, and column header data for view widgets. `Sprox` is also

customizable, so we can go in and modify the way we want our data displayed once we get going with it. Here we define a table widget using Sprox's `sprox.tablebase.TableBase` class for our movie table.:

```
from sprox.tablebase import TableBase

class MovieTable(TableBase):
    __model__ = Movie
    __omit_fields__ = ['genre_id']
movie_table = MovieTable(DBSession)
```

Filling Our Table With Data

So, now we have our `movie_table`, but it's not going to do us much good without data to fill it. Sprox provides a `sprox.fillerbase.TableFiller` class which will retrieve the relevant data from the database and package it in a dictionary for consumption. This is useful if you are creating JSON. Basically, you can provide `CrudRestController` with any object that has a `get_value` function and it will work because of duck typing. Just make certain that your `get_value` function returns the right data type for the widget you are filling. Here is what the filler would look like instantiated.:

```
from sprox.fillerbase import TableFiller

class MovieTableFiller(TableFiller):
    __model__ = Movie
movie_table_filler = MovieTableFiller(DBSession)
```

Putting It All Together

Let's modify our `CrudRestController` to utilize our new table. The new `RootController` would look like this:

```
from tgext.crud import CrudRestController
from moviedemo.model import DBSession, Movie
from sprox.tablebase import TableBase
from sprox.fillerbase import TableFiller

class MovieTable(TableBase):
    __model__ = Movie
movie_table = MovieTable(DBSession)

class MovieTableFiller(TableFiller):
    __model__ = Movie
movie_table_filler = MovieTableFiller(DBSession)

class MovieController(CrudRestController):
    model = Movie
    table = movie_table
    table_filler = movie_table_filler

class RootController(BaseController):
    movie = MovieController(DBSession)
```

You can now visit `/movies/` and it will display a list of movies.

Forms

One of the nice thing about Sprox table definitions is that they provide you with a set of RESTful links. `CrudRestController` provides methods for these pages, but you must provide the widgets for the forms. Specifically, we are talking about the edit and new forms. Here is one way you might create a form to add a new record to the database using `sprox.formbase.AddRecordForm`:

```
class MovieAddForm(AddRecordForm):
    __model__ = Movie
    __omit_fields__ = ['genre_id', 'movie_id']
movie_add_form = MovieAddForm(DBSession)
```

Adding this to your movie controller would look make it now look something like this:

```
class MovieController(CrudRestController):
    model = Movie
    table = movie_table
    table_filler = movie_table_filler
    new_form = movie_add_form
```

You can now visit `/movies/new`.

Edit Form Now we just need to map a form to the edit function so that we can close the loop on our controller. The reason we need separate forms for Add and Edit is due to validation. Sprox will check the database for uniqueness on a “new” form. On an edit form, this is not required since we are updating, not creating.:

```
from sprox.formbase import EditableForm

class MovieEditForm(EditableForm):
    __model__ = Movie
    __omit_fields__ = ['genre_id', 'movie_id']
movie_edit_form = MovieEditForm(DBSession)
```

The biggest difference between this form and that of the “new” form is that we have to get data from the database to fill in the form. Here is how we use `sprox.formbase.EditFormFiller` to do that:

```
from sprox.fillerbase import EditFormFiller

class MovieEditFiller(EditFormFiller):
    __model__ = Movie
movie_edit_filler = MovieEditFiller(DBSession)
```

Now it is a simple as adding our filler and form definitions to the `MovieController` and close the loop on our presentation.

Declarative

If you are interested in brevity, the crud controller may be created in a more declarative manner like this:

```
from ttext.crud import CrudRestController
from sprox.tablebase import TableBase
from sprox.formbase import EditableForm, AddRecordForm
from sprox.fillerbase import TableFiller, EditFormFiller

class DeclarativeMovieController(CrudRestController):
    model = Movie
```

```

class new_form_type(AddRecordForm):
    __model__ = Movie
    __omit_fields__ = ['genre_id', 'movie_id']

class edit_form_type(EditableForm):
    __model__ = Movie
    __omit_fields__ = ['genre_id', 'movie_id']

class edit_filler_type(EditFormFiller):
    __model__ = Movie

class table_type(TableBase):
    __model__ = Movie
    __omit_fields__ = ['genre_id', 'movie_id']

class table_filler_type(TableFiller):
    __model__ = Movie

```

Options reference

The `tgext.crud.CrudRestController` and `tgext.crud.EasyCrudRestController` provide a bunch of configuration options that can be changed by subclassing the controller and providing them in a declarative way:

```
class tgext.crud.CrudRestController(session, menu_items=None)
```

Initialization options

session database session

menu_items Dictionary or mapping type of links to other CRUD sections. This is used to generate the links sidebar of the CRUD interface. Can be specified in the form `model_items['lower_model_name'] = ModelClass` or `model_items['link'] = 'Name'`.

Class attributes

title Title to be used for each page. default: `'Turbogears Admin System'`

model Model this class is associated with.

remember_values List of model attributes that need to keep previous value when not provided on submission instead of replacing the existing value with an empty one. It's commonly used with file fields to avoid having to reupload the file again when the model is edited.

keep_params List of URL parameters that need to be kept around when redirecting between the various pages of the CRUD controller. Can be used to keep around filters or sorting when editing a subset of the models.

search_fields Enables searching on some fields, can be `True`, `False` or a list of fields for which searching should be enabled.

substring_filters Enable substring filtering for some fields, by default is disabled. Pass `True` to enable it on all fields or pass a list of field names to enable it only on some fields.

json_dictify `True` or `False`, enables advanced dictification of retrieved models when providing JSON responses. This also enables JSON encoding of related entities for the returned model.

conditional_update_field Name of the field used to perform conditional updates when `PUT` method is used as a REST API. `None` disables conditional updates (which is the default).

pagination Dictionary of options for pagination. `False` disables pagination. By default `{'items_per_page': 7}` is provided. Currently the only supported option is `items_per_page`.

response_type Limit response to a single format, can be: `'application/json'` or `'text/html'`. By default `tgext.crud` will detect expected response from `Accept` header and will provide response content according to the expected one. If you want to avoid HTML access to a plain JSON API you can use this option to limit valid responses to `application/json`.

resources A list of `CSSSource` / `JSSource` that have to be injected inside CRUD pages when rendering. By default `tgext.crud.resources.crud_style` and `tgext.crud.resources.crud_script` are injected.

table The `sprox.tablebase.TableBase` Widget instance used to display the table. By default `tgext.crud.utils.SortableTableBase` is used which permits to sort table by columns.

table_filler The `sprox.fillerbase.TableFiller` instance used to retrieve data for the table. If you want to customize how data is retrieved override the `TableFiller._do_get_provider_count_and_objs` method to return different entities and count. By default `tgext.crud.utils.RequestLocalTableFiller` is used which keeps track of the number of entities retrieved during the current request to enable pagination.

edit_form Form to be used for editing an existing model. By default `sprox.formbase.EditForm` is used.

edit_filler `sprox.fillerbase.RecordFiller` subclass used to load the values for an entity that need to be edited. Override the `RecordFiller.get_value` method to provide custom values.

new_form Form that defines how to create a new model. By default `sprox.formbase.AddRecordForm` is used.

Customizing Crud Operations

We have really been focusing on the View portion of our controller. This is because `CrudRestController` performs all of the applicable creates, updates, and deletes on your target object for you. This default functionality is provided by `sprox.saormprovider.SAORMProvider`. This can of course be overridden.

Overriding Crud Operations

`CrudRestController` extends `RestController`, which means that any methods available through `RestController` are also available to CRC.

Method	Description	Example Method(s) / URL(s)
get_all	Display the table widget and its data	GET /movies/
new	Display new_form	GET /movies/new
edit	Display edit_form and the containing record's data	GET /movies/1/edit
post	Create a new record	POST /movies/
put	Update an existing record	POST /movies/1?_method=PUT
post_delete	Delete an existing record	POST /movies/1?_method=DELETE
get_delete	Delete Confirmation page	Get /movies/1/delete

If you are familiar with RestController you may notice that `get_one` is missing. There are plans to add this functionality in the near future. Also, you may note the `?_method` on some of the URLs. This is basically a hack because existing browsers do not support the PUT and DELETE methods. Just note that if you decide to incorporate a TW in your `edit_form` description you must provide a `HiddenField('_method')` in the definition.

Adding Functionality

REST provides consistency across Controller classes and makes it easy to override the functionality of a given RESTful method. For instance, you may want to get an email any time someone adds a movie. Here is what your new controller code would look like:

```
class MovieController(CrudRestController):

    # (...)

    @expose(inherit=True)
    def post(self, **kw):
        email_info()
        return super(MovieController, self).post(**kw)
```

You might notice that the function has the `@expose` decorator. This is required because the expose decoration occurs at the class-level, so that means that when you override the class method, the expose is eliminated. We add it back to the method by adding `@expose` with the `inherit` parameter to inherit the behavior from the parent method.

For more details you can refer to the *TGController Subclassing* documentation.

Overriding Templates

To override the template for a given method, you would simple re-define that method, providing an expose to your own template, while simply returning the value of the super class's method.:

```
class MovieController(CrudRestController):

    # (...)

    @expose('movie_demo.templates.my_get_all_template', inherit=True)
    def get_all(self, *args, **kw):
        return super(MovieController, self).get_all(*args, **kw)
```

Removing Functionality

You can also block-out capabilities of the RestController you do not wish implemented. Simply define the function that you want to block, but do not expose it. Here is how we “delete” the delete functionality.:

```
class MovieController(CrudRestController):  
  
    # (...)  
  
    def post_delete(self, *args, **kw):  
        """This is not allowed."""  
        pass
```

Menu Items

The default templates for `tgext.crud` make it very easy to add a menu with links to other resources. Simply provide a dictionary of names and their representing model classes and it will display these links on the left hand side. Here is how you would provide links for your entire model.:

```
import inspect  
from sqlalchemy.orm import class_mapper  
  
models = {}  
for m in dir(model):  
    m = getattr(model, m)  
    if not inspect.isclass(m):  
        continue  
    try:  
        mapper = class_mapper(m)  
        models[m.__name__.lower()] = m  
    except:  
        pass  
  
class RootController(BaseController):  
    movie = MovieController(DBSession, menu_items=models)
```

4.2.5 Working with the TurboGears Admin

TurboGears provides the `tgext.admin` extension which is powered by `tgext.crud` and `sprox`. This can be used to automatically create simple administration pages and is the toolkit powering the `/admin` page in newly quickstarted applications.

By default the admin will provide autogenerated access to all the models imported in your project `models/__init__.py`.

While the default configuration for the admin usually *just works* it doesn't provide the best experience for users and might require some customizations.

To do so the TurboGears Admin permits to provide custom configurations through the `AdminController` `config_type` paramter.

The default turbogears admin is created as:

```
admin = AdminController(model, DBSession, config_type=TGAdminConfig)
```

which creates an admin for all the models with the default TurboGears admin configuration.

Restricting Access to some Models

Restricting access to some models is possible by specifying them explicitly instead of passing `model` as the first argument to the `AdminController`:

```
from myproject.model import User, Group

admin = AdminController([User, Group], DBSession, config_type=TGAdminConfig)
```

Switching to a Custom Admin Configuration

First step to perform if you want to switch to a customized administration configuration is to declare your own config. This can easily be done by subclassing `TGAdminConfig`:

```
class CustomAdminConfig(TGAdminConfig):
    pass
```

if you replace `config_type=TGAdminConfig` with your new `CustomAdminConfig`:

```
admin = AdminController(model, DBSession, config_type=CustomAdminConfig)
```

You will notice that everything works as before, as we didn't change the configuration in any way. The custom configuration will contains properties for the admin itself and for each CRUD inside the admin.

Apart from configuration for each crud, there are some general admin configuration options:

```
class tgext.admin.config.AdminConfig(models, translations=None)
```

Class attributes

layout The look and feel of the admin. Three builtin layouts are available: `tgext.admin.layouts.BasicAdminLayout`, `tgext.admin.layouts.BootstrapAdminLayout` and `tgext.admin.layouts.GroupedBootstrapAdminLayout`.

Note: `GroupedBootstrapAdminLayout` only works with the Genshi template language, so it cannot be used when quickstarting with Jinja or Mako.

default_index_template This is the template of the `/admin` page, by default the one specified inside the layout is used.

allow_only Predicate to restrict access to the whole admin. By default `in_group('managers')` is used.

include_left_menu `bool` that states if the sidebar should be included in admin pages or not. By default the sidebar is visible, please note that hiding the sidebar might break the admin layout.

DefaultControllerConfig The default CRUD configuration for models that do not provide a custom CRUD configuration. By default `tgext.admin.config.CrudRestControllerConfig`

Customizing Models CRUD

The admin page can be configured using the `TGAdminConfig` class, supposing we have a game with running `Match` and a list of `Settings` we can declared `MatchAdminController` and `SettingAdminController` which inherit from `EasyCrudRestController` and tell TurboGears Admin to use them for the administration of matches and settings:

```
class CustomAdminConfig(TGAdminConfig):
    class match(CrudRestControllerConfig):
        defaultCrudRestController = MatchAdminController
```

```
class setting(CrudRestControllerConfig):
    defaultCrudRestController = SettingAdminController
```

```
class RootController(BaseController):
    admin = AdminController([model.Match, model.Setting], DBSession,
                           config_type=CustomAdminConfig)
```

This will create an administration controller which uses our custom `CrudRestController`s to manage `Match` and `Settings` instances.

Each class attribute of our `CustomAdminConfig` that has the *lower case* name of a Model will be used to configure the admin CRUD for that model.

This can be done by subclassing `CrudRestControllerConfig` with a `defaultCrudRestController` class attribute that points to the `CrudRestController` class to use for the related model.

Defining Custom Controllers

The explicit way to define models configuration is by declaring `CrudRestController`s and configurations separately, but in case the custom controllers are required only for the admin it might be shortest to just define them together:

```
class MyModelAdminCrudConfig(CrudRestControllerConfig):
    class defaultCrudRestController(EasyCrudRestController):
        __table_options__ = {
            # options here...
        }
```

```
class CustomAdminConfig(TGAdminConfig):
    photo = MyModelAdminCrudConfig
```

For a complete list of options available inside `defaultCrudRestController` refer to *TurboGears Automatic CRUD Generation*.

Photos Admin Tutorial

Now suppose we have a photo model that looks like:

```
class Photo(DeclarativeBase):
    __tablename__ = 'photos'

    uid = Column(Integer, primary_key=True)
    title = Column(Unicode(255), nullable=False)
    image = Column(LargeBinary, nullable=False)
    mimetype = Column(Unicode(64), nullable=False, default='image/jpeg')

    user_id = Column(Integer, ForeignKey('tg_user.user_id'), index=True)
    user = relationship('User', uselist=False,
                      backref=backref('photos',
                                       cascade='all, delete-orphan'))
```

we might want to customize our admin to use the `GroupedBootstrapAdminLayout` layout, put the photos inside the `Media` group and improve the table view for the crud by removing the `user_id` and `mimetype` columns and declare the `image` column as `xml` so that we can show the image inside:

```

import base64
from tgext.admin import CrudRestControllerConfig
from tgext.admin.tgadminconfig import BootstrapTGAdminConfig as TGAdminConfig
from tgext.admin.layouts import GroupedBootstrapAdminLayout
from tgext.admin.controller import AdminController as TGAdminController
from tgext.crud import EasyCrudRestController

class PhotoAdminCrudConfig(CrudRestControllerConfig):
    icon_class = 'glyphicon-picture'
    admin_group = 'Media'

    class defaultCrudRestController(EasyCrudRestController):
        __table_options__ = {
            '__omit_fields__': ['user_id', 'mimetype'],
            '__xml_fields__': ['image'],

            'image': lambda filler, row: '' % (
                row.mimetype,
                base64.b64encode(row.image).decode('ascii')
            )
        }

class CustomAdminConfig(TGAdminConfig):
    layout = GroupedBootstrapAdminLayout

    photo = PhotoAdminCrudConfig

```

Now our admin works as expected and uses the `PhotoAdminCrudConfig` to manage photos. There is still an open issue, when uploading photos we have to manually provide the photo user and mimetype.

To solve this issue we will customize the form hiding the two values from the form and forcing them when data is submitted:

```

import base64, imghdr
from tg import expose, request
from tgext.admin import CrudRestControllerConfig
from tgext.admin.tgadminconfig import BootstrapTGAdminConfig as TGAdminConfig
from tgext.admin.layouts import GroupedBootstrapAdminLayout
from tgext.admin.controller import AdminController as TGAdminController
from tgext.crud import EasyCrudRestController

class PhotoAdminCrudConfig(CrudRestControllerConfig):
    icon_class = 'glyphicon-picture'
    admin_group = 'Media'

    class defaultCrudRestController(EasyCrudRestController):
        __table_options__ = {
            '__omit_fields__': ['user_id', 'mimetype'],
            '__xml_fields__': ['image'],

            'image': lambda filler, row: '' % (
                row.mimetype,
                base64.b64encode(row.image).decode('ascii')
            )
        }

```

```
__form_options__ = {
    '__hide_fields__': ['user', 'mimetype']
}

@expose(inherit=True)
def post(self, *args, **kw):
    kw['user'] = request.identity['user'].user_id
    kw['mimetype'] = 'image/%s' % imghdr.what(kw['image'].file)
    return EasyCrudRestController.post(self, *args, **kw)

@expose(inherit=True)
def put(self, *args, **kw):
    image = kw.get('image')
    if image is not None and image.filename:
        kw['mimetype'] = 'image/%s' % imghdr.what(kw['image'].file)
    return EasyCrudRestController.put(self, *args, **kw)
```

```
class CustomAdminConfig(TGAdminConfig):
    layout = GroupedBootstrapAdminLayout

    photo = PhotoAdminCrudConfig
```

4.2.6 Rapid Prototyping REST API

TurboGears provides a great way to rapidly prototype rest APIS using the *EasyCrudRestController*.

Defining a basic ready to use API is as simple as:

```
from tgext.crud import EasyCrudRestController

class APIController(EasyCrudRestController):
    pagination = False
    model = model.Permission

class RootController(BaseController):
    permissions = APIController(model.DBSession)
```

Accessing the `/permissions.json` URL you should get:

```
{
  value_list: [
    {
      permission_id: 1,
      description: "This permission give an administrative right to the bearer",
      permission_name: "manage"
    }
  ]
}
```

Blocking non JSON requests

The first side effect is that accessing the url without the `.json` extension you will get the CRUD page. This is usually something you don't want when exposing an API and can be easily prevented by blocking requests which are not in JSON format:

```

from tgext.crud import EasyCrudRestController
from tg import abort

class APIController(EasyCrudRestController):
    pagination = False
    model = model.Permission

    def _before(self, *args, **kw):
        if request.response_type != 'application/json':
            abort(406, 'Only JSON requests are supported')

        super(APIController, self)._before(*args, **kw)

```

Accessing the HTML page will now report a *Not Acceptable* error while JSON output will still be accepted.

Getting Relationships

You probably noticed that even though our permissions are related to groups we didn't get the list of groups the permission is related to.

This is due to the fact that for performance reasons the `EasyCrudRestController` doesn't automatically resolve relationships when providing responses for APIs.

To enable this behavior it is sufficient to turn on the `json_dictify` option:

```

class APIController(EasyCrudRestController):
    pagination = False
    json_dictify = True
    model = model.Permission

    def _before(self, *args, **kw):
        if request.response_type != 'application/json':
            abort(406, 'Only JSON requests are supported')

        super(APIController, self)._before(*args, **kw)

```

Reloading the `/permissions.json` page will now provide the list of groups each `Permission` is related to with a response like:

```

{
  value_list: [
    {
      permission_id: 1,
      description: "This permission give an administrative right to the bearer",
      groups: [
        1
      ],
      permission_name: "manage"
    }
  ]
}

```

Leveraging your REST API on AngularJS

One of the main reasons to rapidly prototype a REST api is to join it with a frontend framework to perform logic and templating on client side.

The following is an example of a working **AngularJS** application that permits creation and deletion of `Permission` objects through the previously created API:

Note: Please note the custom `$resource` to adapted `tgext.crud` responses to the style expected by AngularJS

Note: Pay attention to the double `$$` required to escape the dollar sign on Genshi

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:py="http://genshi.edgewall.org/"
      xmlns:xi="http://www.w3.org/2001/XInclude">

  <xi:include href="master.html" />

<head>
  <title>AngularTG</title>
  <script src="//ajax.googleapis.com/ajax/libs/angularjs/1.2.12/angular.min.js"></script>
  <script src="//ajax.googleapis.com/ajax/libs/angularjs/1.2.12/angular-resource.min.js"></script>
  <style>
    .ng-cloak {
      display: none !important;
    }
  </style>
</head>

<body>
  <div class="row">
    <div class="col-md-12">
      <div ng-app="myApp" class="ng-cloak">
        <div ng-controller="PermissionsCtrl">
          <h1>Permissions</h1>

          <form ng-submit="addPermission(newPerm)">
            <input placeholder="Create Permission" ng-model="newPerm.permission_name" autofocus="true" />
          </form>

          <div ng-repeat="perm in permissions">
            <span ng-click="delPermission($$index)">X</span>
            {{perm.permission_name}}
            <p>{{perm.description}}</p>
          </div>
        </div>
      </div>
    </div>
  </div>
</body>
</script>
<<![CDATA[
  var myApp = angular.module('myApp', ['ngResource']);

  myApp.factory('Permissions', ['$resource', function($resource) {
    return $resource("${tg.url('/permissions/:id.json')}", {'id': "@permission_id"}, {
      query: {
        method: 'GET',
        isArray: true,
        transformResponse: function (data) {
          var data = angular.fromJson(data);
          return data.value_list;
        }
      }
    });
  ]]
```

```

    },
    save: {
        method: 'POST',
        transformResponse: function(data) {
            var data = angular.fromJson(data);
            return data.value;
        }
    }
    });
}));

myApp.controller('PermissionsCtrl', ['$rootScope', 'Permissions',
    function ($rootScope, Permissions) {
        $rootScope.permissions = Permissions.query();

        $rootScope.addPermission = function(permData) {
            var perm = new Permissions(permData);
            permData.permission_name = "";
            perm.$$save(function(data) {
                $rootScope.permissions.push(new Permissions(data));
            });
        }

        $rootScope.delPermission = function(index) {
            perm = $rootScope.permissions[index];
            perm.$$delete(function(data) {
                $rootScope.permissions.splice(index, 1);
            });
        }
    }
}]);
</script>
</body>
</html>

```

Limiting API Results

The previous API returns all the Permissions available, which is the most simple case but not always what you are looking for. It is often needed to filter the results by a constraint, for example it is common to get only the objects for a specific user.

While this can be easily achieved by passing any filter to the API itself when it is called: `/permissions.json?groups=1`. It is common to need to perform this on server side.

If we want to permanently only get the permissions for the manage groups we can make it by extending the `get_all` method:

```

from tgext.crud import EasyCrudRestController
from tg import abort

class APIController(EasyCrudRestController):
    pagination = False
    json_dictify = True
    model = model.Permission

    def _before(self, *args, **kw):
        if request.response_type != 'application/json':

```

```
        abort(406, 'Only JSON requests are supported')

    super(APIController, self)._before(*args, **kw)

    @expose(inherit=True)
    def get_all(self, *args, **kw):
        kw['groups'] = 1
        return super(APIController, self).get_all(*args, **kw)
```

If you point your browser to `/permissions.json` and had multiple permissions you will see that only those for the `managers` group are now reported.

Now if you tried to use the filtered controller with the previously created **AngularJS** application you probably noticed that the new permissions you create are not listed back when you reload the page. This is because they are actually created without a group, so they don't match our `managers` group filter.

To avoid this we can also force the group on creation by extending also the `post` method:

```
from tgext.crud import EasyCrudRestController
from tg import abort

class ApiController(EasyCrudRestController):
    pagination = False
    json_dictify = True
    model = model.Permission

    def _before(self, *args, **kw):
        if request.response_type != 'application/json':
            abort(406, 'Only JSON requests are supported')

        super(APIController, self)._before(*args, **kw)

    @expose(inherit=True)
    def get_all(self, *args, **kw):
        kw['groups'] = 1
        return super(APIController, self).get_all(*args, **kw)

    @expose(inherit=True)
    def post(self, *args, **kw):
        kw['groups'] = [1]
        return super(APIController, self).post(*args, **kw)
```

This will now correctly create all the new permissions for the `managers` group.

Custom Queries when fetching data

While extending the `get_all` method is quick and easy, you are limited to the filtering possibilities that **sprox** exposes you.

For more advanced filtering or even custom queries it is possible to declare your own `TableFiller` with a totally custom query:

```
from tgext.crud import EasyCrudRestController
from sprox.fillerbase import TableFiller
from tg import abort

class ApiController(EasyCrudRestController):
    pagination = False
```

```

json_dictify = True
model = model.Permission

class table_filler_type(TableFiller):
    __entity__ = model.Permission

    def _do_get_provider_count_and_objs(self, **kw):
        manager_group = model.DBSession.query(model.Group).filter_by(group_name='managers').first
        results = model.DBSession.query(model.Permission).filter(model.Permission.groups.contains
        return len(results), results

    def _before(self, *args, **kw):
        if request.response_type != 'application/json':
            abort(406, 'Only JSON requests are supported')

        super(APIController, self)._before(*args, **kw)

```

4.2.7 SQLAlchemy Master Slave Load Balancing

Since version 2.2 TurboGears has basic support for Master/Slave load balancing and provides a set of utilities to use it.

TurboGears permits to declare a master server and any number of slave servers, all the writes will automatically redirected to the master node, while the other calls will be dispatched randomly to the slave nodes.

All the queries executed outside of TurboGears controllers will run only on the master node, those include the queries performed by the authentication stack to initially look up an already logged in user, its groups and permissions.

Enabling Master Slave Balancing

To enable Master Slave load Balancing you just need to edit your *model/__init__.py* making the *sessionmaker* use the TurboGears *BalancedSession*:

```

from tg.configuration.sqla.balanced_session import BalancedSession

maker = sessionmaker(autoflush=True, autocommit=False,
                    class_=BalancedSession,
                    extension=ZopeTransactionExtension())

```

Doing this by itself will suffice to make load balancing work, but still as there is only the standard database configuration the *BalancedSession* will just be redirecting all the queries to the only available serve.

Configuring Balanced Nodes

To let load balancing work we must specify at least a master and slave server inside our application configuration. The master server can be specified using the *sqlalchemy.master* set of options, while any number of slaves can be configured using the *sqlalchemy.slaves* options:

```

sqlalchemy.master.url = mysql://username:password@masterhost:port/databasename
sqlalchemy.master.pool_recycle = 3600

sqlalchemy.slaves.slave1.url = mysql://username:password@slavehost:port/databasename
sqlalchemy.slaves.slave1.pool_recycle = 3600

```

The master node can be configured also to be a slave, this is usually the case when we want the master to also handle some read queries.

Driving the balancer

TurboGears provides a set of utilities to let you change the default behavior of the load balancer. Those include the `@with_engine(engine_name)` decorator and the `DBSession().using_engine(engine_name)` context.

The with_engine decorator

The `with_engine` decorator permits to force a controller method to run on a specific node. It is a great tool for ensuring that some actions take place on the master node, like controllers that edit content.

```
from tg import with_engine

@expose('myproj.templates.about')
@with_engine('master')
def about(self):
    DBSession.query(model.User).all()
    return dict(page='about')
```

The previous query will be executed on the master node, if the `@with_engine` decorator is removed it will get executed on any random slave.

The `with_engine` decorator can also be used to force turbogears to use the master node when some parameters are passed by url:

```
@expose('myproj.templates.index')
@with_engine(master_params=['m'])
def index(self):
    DBSession.query(model.User).all()
    return dict(page='index')
```

In this case calling `http://localhost:8080/index` will result in queries performed on a slave node, while calling `http://localhost:8080/index?m=1` will force the queries to be executed on the master node.

Pay attention that the `m=1` parameter can actually have any value, it just has to be there. This is especially useful when redirecting after an action that just created a new item to a page that has to show the new item. Using a parameter specified in `master_params` we can force TurboGears to fetch the items from the master node so to avoid odd results due to data propagation delay.

Keeping master_params around By default parameters specified in `with_engine` `master_params` will be popped from the controller params. This is to avoid messing with validators or controller code that doesn't expect the parameter to exist.

If the controller actually needs to access the parameter a dictionary can be passed to `@with_engine` instead of a list. The dictionary keys will be the parameters, while the value will be if to pop it from the parameters or not.

```
@expose('myproj.templates.index')
@with_engine(master_params={'m':False})
def index(self, m=None):
    DBSession.query(model.User).all()
    return dict(page='index', m=m)
```

Forcing Single Queries on a node

Single queries can be forced to execute on a specific node using the `using_engine` method of the `BalancedSession`. This method returns a context manager, until queries are executed inside this context they are run on the constrained engine:

```
with DBSession().using_engine('master'):
    DBSession.query(model.User).all()
    DBSession.query(model.Permission).all()
DBSession.query(model.Group).all()
```

In the previous example the Users and the Permissions will be fetched from the master node, while the Groups will be fetched from a random slave node.

Debugging Balancing

Setting the root logger of your application to `DEBUG` will let you see which node has been chosen by the `BalancedSession` to perform a specific query.

4.2.8 Deploying TurboGears

This section describes standard deployment techniques for TurboGears2.

Running TurboGears under Apache with `mod_wsgi`

`mod_wsgi` is an Apache module developed by Graham Dumpleton. It allows WSGI programs to be served using the Apache web server.

This guide will outline broad steps that can be used to get a TurboGears application running under Apache via `mod_wsgi`.

1. The tutorial assumes you have Apache already installed on your system. If you do not, install Apache 2.X for your platform in whatever manner makes sense.
2. Once you have Apache installed, install `mod_wsgi`. Use the (excellent) [installation instructions](#) for your platform into your system's Apache installation.
3. Create a virtualenvironment with the specific TurboGears version your application depends on installed.

```
$ virtualenv /var/tg2env
$ /var/tg2env/bin/pip install tg.devtools
```

4. Activate the virtualenvironment

```
$ source /var/tg2env/bin/activate
(tg2env)$ #virtualenv now activated
```

5. Install your TurboGears application.

```
(tg2env)$ cd /var/www/myapp
(tg2env)$ python setup.py develop
```

6. Within the application director, create a script named `app.wsgi`. Give it these contents:

```
APP_CONFIG = "/var/www/myapp/myapp/production.ini"

#Setup logging
import logging.config
logging.config.fileConfig(APP_CONFIG)

#Load the application
from paste.deploy import loadapp
application = loadapp('config:%s' % APP_CONFIG)
```

7. Edit your Apache configuration and add some stuff.

```
<VirtualHost *:80>
    ServerName www.site1.com

    WSGIProcessGroup www.site1.com
    WSGIDaemonProcess www.site1.com user=www-data group=www-data threads=4 python-path=/var/tg2
    WSGIScriptAlias / /var/www/myapp/app.wsgi

    #Serve static files directly without TurboGears
    Alias /images /var/www/myapp/myapp/public/images
    Alias /css /var/www/myapp/myapp/public/css
    Alias /js /var/www/myapp/myapp/public/js

    CustomLog logs/www.site1.com-access_log common
    ErrorLog logs/www.site1.com-error_log
</VirtualHost>
```

8. Restart Apache

```
$ sudo apache2ctl restart
```

9. Visit <http://www.site1.com/> in a browser to access the application.

See the [mod_wsgi configuration documentation](#) for more in-depth configuration information.

Running TurboGears under Circus and Chaussette

Circus is a process & socket manager. It can be used to monitor and control processes and sockets, when paired with the Chaussette WSGI server it can become a powerful tool to deploy your application and manage any related process your applications needs.

Circus can take care of starting your memcached, redis, database server or batch process with your application itself providing a single point where to configure the full application environment.

This guide will outline broad steps that can be used to get a TurboGears application running under Chaussette through Circus.

1. The tutorial assumes you have Circus already installed on your system. If you do not, install it in whatever manner makes sense for your environment.

A possible way is by performing:

```
$ pip install circus
```

2. Create a virtual environment with the specific TurboGears version your application depends on installed.

```
$ virtualenv /var/tg2env
$ /var/tg2env/bin/pip install tg.devtools
```

3. Activate the virtual environment

```
$ source /var/tg2env/bin/activate
(tg2env)$ #virtualenv now activated
```

4. Once you have the environment enabled you will need to install the Chaussette WSGI Server:

```
(tg2env)$ pip install chaussette
```

5. Chaussette supports many backends to serve the requests. The default one is based on wsgiref, which is not really fast. Have a look at the [Chaussette Documentation](#) for the available backends: waitress, gevent, meinheld and many more are supported.

For this tutorial we are going to use Waitress, which is a multithreaded WSGI server, so we need to install it inside virtual environment:

```
(tg2env)$ pip install waitress
```

6. Now the environment is ready for deploy, you just need to install the TurboGears application.

```
(tg2env)$ cd /var/www/myapp
(tg2env)$ python setup.py develop
```

7. We now create a circus configuration file (named `circus.ini`) with the informations required to load and start your application. This can be performed using the `gearbox deploy-circus` command from `gearbox-tools` package or by manually writing it:

```
[circus]
check_delay = 5
endpoint = tcp://127.0.0.1:5555
debug = true

[env:myapp]
PATH=/var/tg2env/bin:$PATH
VIRTUAL_ENV=/var/tg2env

[watcher:myapp]
working_dir = /var/www/myapp
cmd = chaussette --backend waitress --fd $(circus.sockets.myapp) paste:production.ini
use_sockets = True
warmup_delay = 0
numprocesses = 1

stderr_stream.class = FileStream
stderr_stream.filename = /var/log/circus/myapp.log
stderr_stream.refresh_time = 0.3

stdout_stream.class = FileStream
stdout_stream.filename = /var/log/circus/myapp.log
stdout_stream.refresh_time = 0.3

[socket:myapp]
host = localhost
port = 8080
```

8. Now start circus with the configuration file, after being started it will load your application:

```
$ circusd circus.ini

2013-02-15 18:19:54 [20923] [INFO] Starting master on pid 20923
```

```
2013-02-15 18:19:54 [20923] [INFO] sockets started
2013-02-15 18:19:54 [20923] [INFO] myapp started
2013-02-15 18:19:54 [20923] [INFO] Arbiter now waiting for commands
```

9. Visit `http://localhost:8080/` in a browser to access the application. You can now proxy it behind Apache, Nginx or any other web server or even use the [VHostino](#) project for circus to serve multiple applications through virtual hosts

See the [circus documentation](#) for more in-depth configuration information.

Running TurboGears under Heroku

This recipe assumes that you have a TurboGears app setup using a Paste INI file, inside a package called 'myapp'. If you are deploying a custom TurboGears application in minimal mode you might have to tune the following instructions.

Step 0: Install heroku

Install the heroku gem [per their instructions](#).

Step 1: Add files needed for heroku

You will need to add the following files with the contents as shown to the root of your project directory (the directory containing the `setup.py`).

```
requirements.txt:
```

You can autogenerate this file by running:

```
$ pip freeze > requirements.txt
```

You will probably have a line in your requirements file that has your project name in it. It might look like either of the following two lines depending on how you setup your project. If either of these lines exist, **delete them**.

```
projectname=0.1dev
or
-e git+git@xxxx:<git username>/xxxxx.git...#egg=projectname
```

Now that you have properly frozen your application dependencies it is required to add the webserver you want to use to actually serve your application requests.

This tutorial uses the *Waitress* webserver, so we need to add it to the dependencies declared in the `requirements.txt`

```
$ echo "waitress" >> requirements.txt
```

Step 2: Editing Configuration File

As heroku passes some configuration options in ENVIRON variables, it is necessary for our application to read them from the Heroku environment. Those are typically the `PORT` where your application server has to listen, the `URL` of your database and so on...

First of all we need to copy the `development.ini` to a `production.ini` file we are going to use for the heroku deployment:

```
$ cp development.ini production.ini
```

The only options you are required to change are the one related to the server. So your `[server:main]` section should look like:

```
[server:main]
use = egg:waitress#main
host = 0.0.0.0
get_port = heroku_port
```

Then probably want to disable the debug inside the `[DEFAULT]` section.

Note: If you want to use a different server instead of waitress, as gevent, CherryPy or something else, as far as it is compatible with PasteDeploy changing the `use = egg:waitress#main` to whatever you want usually is enough. In case you want to use gevent, for you can change it to `use = egg:gearbox#gevent`.

Step 3: Starting the application

Procfile:

Generate this by running:

```
$ echo "web: ./run" > Procfile
```

run:

Create run with the following:

```
#!/bin/bash
python setup.py develop
gearbox serve --debug -c production.ini heroku_port=$PORT
```

Note: Make sure to `chmod +x run` before continuing. The ‘develop’ step is necessary because the current package must be installed before paste can load it from the INI file.

Step 4: Setup git repo and heroku app

Navigate to your project directory (directory with `setup.py`) if not already there. If you project is already under git version control, skip to the ‘Initialize the heroku stack’ section.

Inside your projects directory, if this project is not tracked under git it is recommended that you first create a good `.gitignore` file (you can skip this step). You can get the recommended python one by running:

```
$ wget -O .gitignore https://raw.githubusercontent.com/github/gitignore/master/TurboGears2.gitignore
```

Once that is done, run:

```
$ git init
$ git add .
$ git commit -m "initial commit"
```

Step 5: Initialize the heroku stack

```
$ heroku create
```

Step 6: Deploy

To deploy a new version, push it to heroku:

```
$ git push heroku master
```

Make sure to start one worker:

```
$ heroku scale web=1
```

Check to see if your app is running

```
$ heroku ps
```

Take a look at the logs to debug any errors if necessary:

```
$ heroku logs -t
```

Running TurboGears under AppEngine

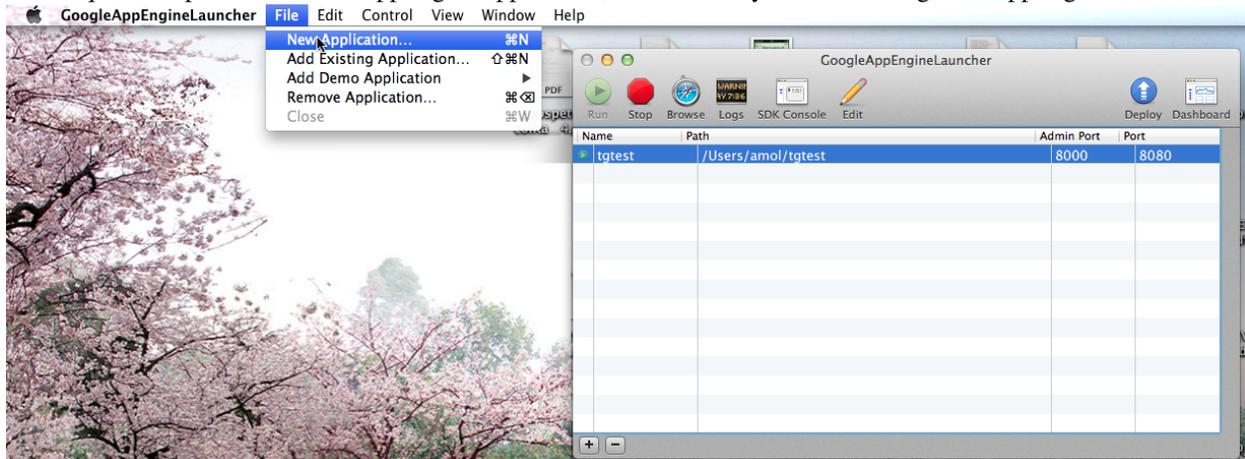
This recipe describes how to create a minimal mode TurboGears application under Google AppEngine.

Step 0: Install AppEngine

Install the AppEngine SDK from https://developers.google.com/appengine/downloads#Google_App_Engine_SDK_for_Python
The following guide uses the AppEngine SDK for OSX.

Step 1: Create Application

First required step is to create an AppEngine application, this can easily be done through the AppEngine SDK.



If you are not using the AppEngine SDK for OSX please refer to the [AppEngine Documentation](#) to create a new application.

This will create a new application in the specified directory with the following files:

```
$ tree
.
-- app.yaml
-- favicon.ico
-- index.yaml
-- main.py
```

By default the created application relies on the Webapp2 framework, to remove this dependency edit the `app.yaml` file and delete the following part:

```
libraries:
- name: webapp2
  version: "2.5.2"
```

Step 2: Setup Dependencies

On AppEngine, all the dependencies of your application should be provided with the application itself. This makes so that we are required to install TurboGears inside the application directory.

Supposing your application is named `tgtest` and you created it in your HOME, to do so we first need to create a temporary virtual environment we will throw away when everything is installed:

```
$ cd ~/tgtest
$ virtualenv --no-site-packages tmpenv
New python executable in tmpenv/bin/python
Installing setuptools.....done.
Installing pip.....done.
```

Note: Depending on your `virtualenv` command version, the `--no-site-packages` option might not be required.

Now will enable the `virtualenv` and install the TurboGears2 dependency, only difference is that instead of installing it inside the virtual environment itself we will tell `pip` to install it inside the `packages` directory of our application:

```
. tmpenv/bin/activate
(tmpenv)$ pip install -t packages -I TurboGears2
```

As AppEngine doesn't provide `setuptools` the last required step is to also provide `setuptools` inside our `packages` directory:

```
(tmpenv)$ pip install -t packages -I setuptools
```

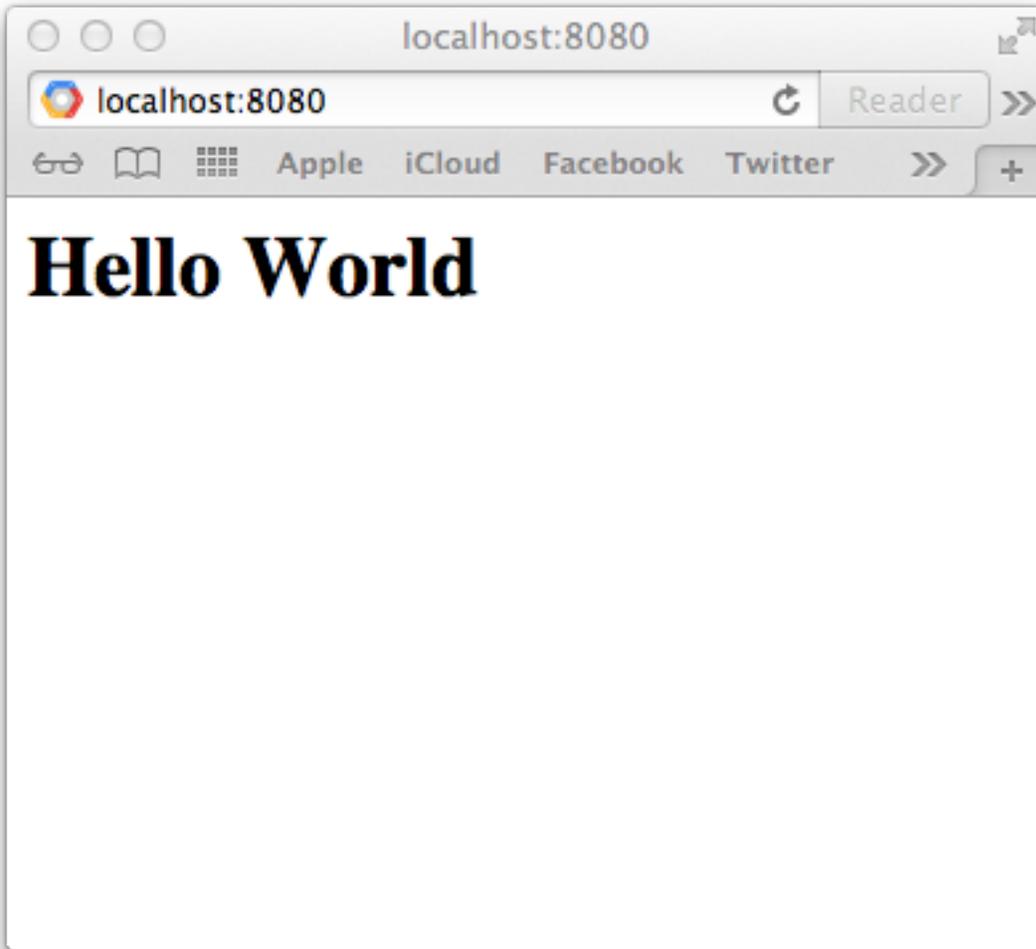
Note: Please note the `-I` option to force `pip` ignoring the currently installed packages.

Now all the required dependencies are installed inside the `packages` directory and our virtual environment can be deleted:

```
(tmpenv)$ deactivate
$ rm -r tmpenv
```

Step 3: The TurboGears Application

Now we can proceed editing the `main.py` file which is started by AppEngine to run our application and actually write a TurboGears application there.



Note: if something went wrong, you can press the `Logs` button and check the application output.

Step 5: Deploy It!

Now that your application is correctly running locally, it's time to send it to Google servers.

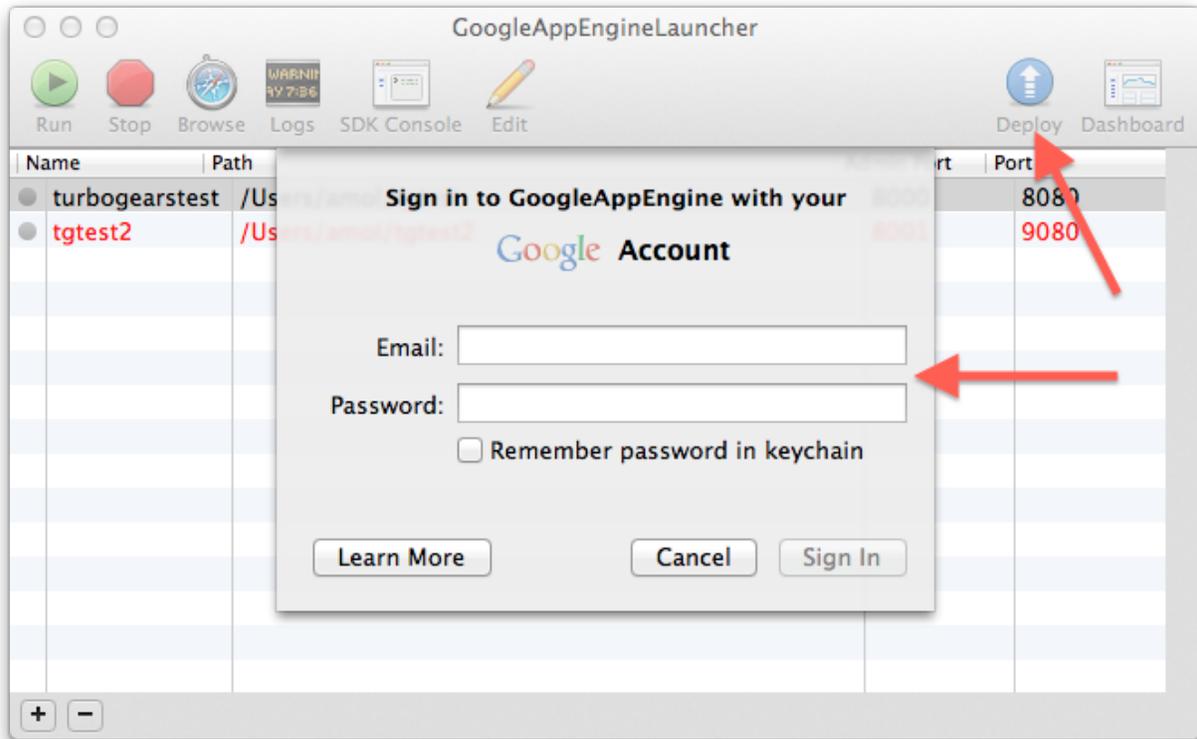
First required step, is to let google know that our application exists. To do so, point your browser to `https://appengine.google.com/` and press the `Create Application` button to create a new application.

Once you created your application and have a valid `Application Identifier` available, edit the `app.yaml` file inside `tgtest` to set the correct identifier:

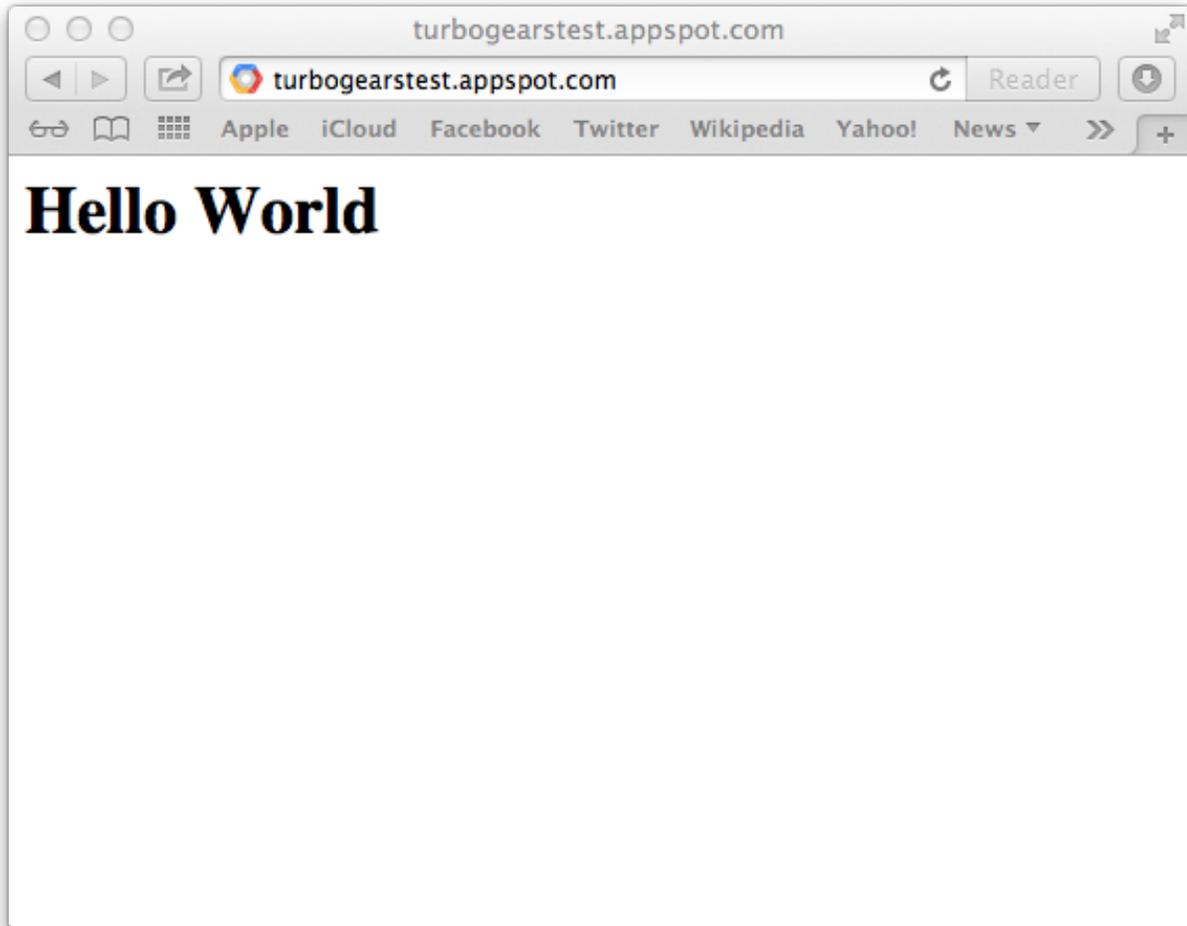
```
application: turbogearstest
```

Note: replace `turbogearstest` with your own app identifier.

Now get back to the `GoogleAppEngineLauncher` and press the **Deploy** button. It will ask for your google account credentials, and will then proceed with the deploy.



As soon as it finished the deploy, your application will be available online at the reserved URL. Which is in the form *APPID.appspot.com*, in our example we can point the browser to `http://turbogearstest.appspot.com` and get our application output.



4.2.9 Setting up logging in your Application

TurboGears relies on the standard Python `logging` module, so to add logging to your application simply add the following lines at the begin of your files:

```
import logging
log = logging.getLogger(__name__)
```

Then you can report logging messages with the standard logger methods: `log.warning()`, `log.info()`, `log.error()`, `log.exception()` and so on:

```
class SimpleController(TGController):

    @expose()
    def simple(self):
        log.debug("My first logged controller!")
        return "OK"
```

Refer to the Python *Logger documentation* for additional details.

By default TurboGears configures the logging module so that all the log messages from your application are displayed from `DEBUG` level on. So even debug messages will be displayed.

This is specified at the end of the `development.ini` file. When starting your application with `gearbox` it will automatically load the logging configuration from your `development.ini` or provided configuration file.

When you are deploying your application on `mod_wsgi` or any other environment that doesn't rely on `gearbox` to run the application, remember to load the logging configuration before creating the actual WSGI application:

```
APP_CONFIG = "/var/www/myapp/myapp/production.ini"

#Setup logging
import logging.config
logging.config.fileConfig(APP_CONFIG)

#Load the application
from paste.deploy import loadapp
application = loadapp('config:%s' % APP_CONFIG)
```

Otherwise the logging configuration will be different from the one available when starting the application with `gearbox` and you might end up not seeing logging messages.

Logging Output

In the default configuration all your logging output goes to `sys.stderr`. What exactly that is depends on your deployment environment.

In case of `mod_wsgi` it will be redirected to the Apache `ErrorLog`, but in case your environment doesn't provide a convenient way to configure output location you can set it up through the `development.ini` in the `[handler_console]` section:

```
[handler_console]
class = StreamHandler
args = (sys.stderr,)
level = NOTSET
formatter = generic
```

For example to change it to log to a specific file you can replace the `StreamHandler` with a `FileHandler`:

```
[handler_console]
class = FileHandler
args = ('application.log', 'a')
level = NOTSET
formatter = generic
```

Note: Please not that the best practice is not to change the `console` handler but creating a new handler and switch the various loggers to it.

WSGI Errors Output

The WSGI standard defines a `wsgi.errors` key in the environment which can be used to report application errors. As this feature is only available during a request (when the WSGI environment is provided), applications won't usually rely on it, preferring instead the logging module which is always available.

Please note that many WSGI middlewares will log to it, instead of using the logging module, such an example is the `backlash` error reporting middleware used by TurboGears for its errorware stack.

Setting up `wsgi.errors` is usually a task that your application server does for you, and will usually point to the same location `sys.stderr` points to. So your `wsgi.errors` and logging outputs should be available at the same destination.

In case your deploy environment isn't setting up `wsgi.errors` correctly or you changed the logging output you might have to change where `wsgi.errors` points too.

This has to be done by code, replacing the `environ['wsgi.errors']` key, on every request, with a stream object. Being it `sys.stderr` or something else.

It is usually best practice to leave both the logging output on `sys.stderr` and `wsgi.errors` as is, as they will usually end up at the same location on most application servers. Then you can tune the output from the application server configuration itself.

In case of `gearbox serve`, `wsgi.errors` will point to `sys.stderr` which is then redirected to a logfile, if provided with the `--log-file` option.

In case of `mod_wsgi` they will both point to the Apache `ErrorLog` file so you can tune your whole logging output configuration from Apache itself.

4.3 Advanced Recipes

4.3.1 Streaming Response

Streaming permits to your controller to send data that yet has to be created to the client, this can be really useful when your app needs to send an huge amount of data to the client, much more data than you are able to keep in memory.

Streaming can also be useful when you have to send content to the client that yet has to be generated when you provide an answer to the client keeping an open connection between the application and the client itself.

In TurboGears2 streaming can be achieved returning a generator from your controllers.

Making your application streaming compliant

So the first thing you want to do when using streaming is disabling any middleware that edits the content of your response. Since version 2.3 disabling debug mode is not required anymore.

Most middlewares like `debugbar`, `ToscaWidgets` and so on will avoid touching your response if it is not of `text/html` content type. So streaming files or json is usually safe.

Streaming with Generators

Streaming involves returning a generator from your controller, this will let the generator create the content while being read by client.

```
@expose(content_type='application/json')
def stream_list(self):
    def output_pause():
        num = 0
        yield '['
        while num < 9:
            num += 1
            yield '%s, ' % num
            time.sleep(1)
        yield '10]'
    return output_pause()
```

This simple example will slowly stream the numbers from 1 to 10 in a json array.

Accessing TurboGears objects

While streaming content is quite simple some teardown functions get executed before your streamer, this has the side effect of changing how your application behaves.

Accessing Request

Since version 2.3 all the global turbogears objects are accessible while running the generator. So if you need to have access to them, you can freely read and write to them. Just keep in mind that your response has already been started, so it is not possible to change your outgoing response while running the generator.

```
@expose(content_type='text/css')
def stream(self):
    def output_pause(req):
        num = 0
        while num < 10:
            num += 1
            yield '%s/%s\n' % (tg.request.path_info, num)
            time.sleep(1)
    return output_pause()
```

This example, while not returning any real css, shows how it is possible to access the turbogears request inside the generator.

Reading from Database

Since version 2.3 it is possible to read from the database as usual:

```
@expose(content_type='application/json')
def stream_db(self):
    def output_pause():
        num = 0
        yield '['
        while num < 9:
            u = DBSession.query(model.User).filter_by(user_id=num).first()
            num += 1
            yield u and '%d, ' % u.user_id or 'null, '
            time.sleep(1)
        yield 'null]'
    return output_pause()
```

Writing to Database

If you need to write data on the database you will have to manually flush the session and commit the transaction. This is due to the fact that TurboGears2 won't be able to do it for you as the request flow already ended.

```
@expose(content_type='application/json')
def stream_list(self):
    def output_pause():
        import transaction
        num = 0
        while num < 9:
            DBSession.add(model.Permission(permission_name='perm_%s' % num))
            yield 'Added Permission\n'
```

```

        num += 1
        time.sleep(1)
        DBSession.flush()
        transaction.commit()
    return output_pause()

```

4.3.2 Scheduling Tasks

On Posix systems, using cron is a perfectly valid way to schedule tasks for your application.

However, it sometimes happen that you need to interact intimately with the runtime environment of your application, that you need to schedule jobs dynamically, or that your hosting service does not provide access to cron. In those cases, you can schedule jobs with the *TGScheduler* module.

Installation

TGScheduler is registered on PyPI and therefore can be installed with *easy_install*:

```
$ pip install tgscheduler
```

Setup

TGScheduler is not started by default. To allow your tasks to run, simply start the scheduler when your application is loaded. You can do that in *lib/app_globals.py*:

```

import tgscheduler

class Globals(object):
    def __init__(self):
        tgscheduler.start_scheduler()

```

Scheduling Tasks

To you have four ways to schudule tasks:

- *add_interval_task()*;
- *add_monthly_task()*;
- *add_single_task()*;
- *add_weekday_task()*.

Each of those receive a callable and a time specifier that defines when to run a function. As an example, if you want to update the stock prices in your database every 15 minutes, you would do something like the following:

```

def update_stocks():
    url = 'http://example.com/stock_prices.xml'
    data = urllib2.urlopen(url).read()
    etree = lxml.etree.fromstring(data)
    for el in etree.xpath("//stock"):
        price = model.StockPrice(el.get("name"), int(el.get("price")))
        model.DBSession.add(price)

```

```

class Globals(object):

```

```
def __init__(self):
    tgscheduler.start_scheduler()
    tgscheduler.add_interval_task(60*15, update_stocks)
```

4.3.3 Advanced Caching Recipes

Caching Template and Controller

A simple form of joint controller+template caching can be achieved by using both `cached` decorator and `tg_cache` parameter as described in *Caching*.

While it is more common having to perform some kind of minimal computation in controller to decide which cache key to use and rely on the template caching only, the caching system can be leveraged to turn the whole request in a direct cache hit based on the request parameters.

This can be achieved by relying on the `cached` decorator and using `render_template()` to actually render the template during controller execution and caching it together with the controller itself:

```
from tg import cached, render_template

@cached()
@expose(content_type='text/html')
def cached_func(self, what='about'):
    return render_template(dict(page=what, time=time.time()),
                           'genshi', 'myproj.templates.cached_func')
```

Note: While `@cached` caches the controller itself, any hook or validation associated to the controller will still be executed. This might be what you want (for example when tracking page views through an hook) or it might not, depending on your needs you might want to move hooks and validation inside the controller itself to ensure they are cached.

Caching Authentication

Authentication in TurboGears is provided by `repoze.who` through the `ApplicationAuthMetadata` class as described in *Authentication in TurboGears 2 applications*.

`ApplicationAuthMetadata` provides two major steps in authentication:

- One is authenticating the user itself for the first time (done by `ApplicationAuthMetadata.authenticate`) which is in charge of returning the `user_name` that uniquely identifies the user (or any other unique identifier) that is then received by the other methods to lookup the actual user data.
- The second step, which is the metadata provisioning, is performed on each request and receives the previously identified user as returned by the authentication step. This is performed by `get_user`, `get_groups` and `get_permissions` which are in charge of returning the three aforementioned information regarding the user.

It's easy to see that this second step is usually the one that has most weight over the request throughput as it involves three different queries to the database. Usually we cannot rely on the TurboGears caching for those methods as they happen before the TurboGears cache is ready, but fortunately we can still cache it by writing an `ApplicationWrapper` that performs the work in place of the metadata provisioning functions.

First action is to change our `ApplicationAuthMetadata` to actually avoid fetching any data:

```

from tg.util import Bunch

class ApplicationAuthMetadata(TGAuthMetadata):
    def __init__(self, sa_auth):
        self.sa_auth = sa_auth

    def authenticate(self, environ, identity):
        ... # authenticate implementation

    def get_user(self, identity, userid):
        return Bunch(user_name=userid)

    def get_groups(self, identity, userid):
        return []

    def get_permissions(self, identity, userid):
        return []

```

```
base_config.sa_auth.authmetadata = ApplicationAuthMetadata(base_config.sa_auth)
```

This will actually do nothing and just return a fake user for requests when they perform the metadata provisioning during authentication.

The real metadata provisioning will be performed by an ad-hoc ApplicationWrapper that can access the cache:

```

from tg.appwrappers.base import ApplicationWrapper

class AuthMetadataApplicationWrapper(ApplicationWrapper):
    def __init__(self, next_handler, config):
        super(AuthMetadataApplicationWrapper, self).__init__(next_handler, config)
        self.sa_auth = config['sa_auth']

    def get_auth_metadata(self, userid):
        user = self.sa_auth.dbsession.query(self.sa_auth.user_class).filter_by(user_name=userid).first()
        return {
            'user': user,
            'groups': [g.group_name for g in user.groups],
            'permissions': [p.permission_name for p in user.permissions]
        }

    def __call__(self, controller, environ, context):
        identity = environ.get('repoze.who.identity')
        if identity is not None:
            userid = identity['repoze.who.userid']
            auth_cache = context.cache.get_cache('auth')
            auth_metadata = auth_cache.get_value(key=userid,
                                                createfunc=lambda: self.get_auth_metadata(userid),
                                                expiretime=3600)

            auth_metadata['user'] = self.sa_auth.dbsession.merge(auth_metadata['user'], load=False)

            identity.update(auth_metadata)
            environ['repoze.what.credentials'].update(identity)

        return self.next_handler(controller, environ, context)

```

```
base_config.register_wrapper(AuthMetadataApplicationWrapper)
```

This is usually enough to cache authentication requests in an environment where user data, permissions and groups

change rarely. A better cache management, invalidating the user cache whenever the user itself or its permission change, is required for more volatile scenarios.

4.3.4 Using Multiple Databases In TurboGears

Status RoughDoc

Table of Contents

- Using Multiple Databases In TurboGears
 - Define your database urls in the [app:main] section of your .ini file(s)
 - Change The Way Your App Loads The Database Engines
 - Update Your Model's `__init__` To Handle Multiple Sessions And Metadata
 - Tell Your Models Which Engine To Use
 - Optional: Create And Populate Each Database In `Websetup.py`

The goal of this tutorial is to configure TurboGears to use multiple databases. In this tutorial we will simply set up two different databases engines that will use db session handlers of `DBSession` and `DBSession2`, db metadata names of `metadata` and `metadata2`, and `DeclarativeBase` objects of `DeclarativeBase` and `DeclarativeBase2`.

Define your database urls in the [app:main] section of your .ini file(s)

The first thing you will need to do is edit your .ini file to specify multiple url options for the sqlalchemy configuration.

In `myapp/development.ini` (or `production.ini`, or whatever.ini you are using), comment out the original sqlalchemy.url assignment and add the multiple config options:

```
#sqlalchemy.url = sqlite:///%(here)s/devdata.db
sqlalchemy.first.url = sqlite:///%(here)s/database_1.db
sqlalchemy.second.url = sqlite:///%(here)s/database_2.db
```

Change The Way Your App Loads The Database Engines

Now we need to instruct the app to load the multiple databases correctly. This requires telling `base_config` (in `app_cfg.py`) to load our own custom `AppConfig` with the proper multi-db assignments and a call to the model's `init_model` method (more on that in the next step).

In `myapp/config/app_cfg.py`:

```
# make sure these imports are added to the top
from tg.configuration import AppConfig, config
from myapp.model import init_model

# add this before base_config =
class MultiDBAppConfig(AppConfig):
    def setup_sqlalchemy(self):
        """Setup SQLAlchemy database engine(s)"""
        from sqlalchemy import engine_from_config
        engine1 = engine_from_config(config, 'sqlalchemy.first.')
        engine2 = engine_from_config(config, 'sqlalchemy.second.')
        # engine1 should be assigned to sa_engine as well as your first engine's name
        config['tg.app_globals'].sa_engine = engine1
        config['tg.app_globals'].sa_engine_first = engine1
        config['tg.app_globals'].sa_engine_second = engine2
```

```

    # Pass the engines to init_model, to be able to introspect tables
    init_model(engine1, engine2)

#base_config = AppConfig()
base_config = MultiDBAppConfig()

```

Update Your Model's `__init__` To Handle Multiple Sessions And Metadata

Switching the model's init from a single-db config to a multi-db simply means we have to duplicate our DBSession and metadata assignments, and then update the `init_model` method to assign/configure each engine correctly.

In `myapp/model/__init__.py`:

```

# after the first maker/DBSession assignment, add a 2nd one
maker2 = sessionmaker(autoflush=True, autocommit=False,
                      extension=ZopeTransactionExtension())
DBSession2 = scoped_session(maker2)

# after the first DeclarativeBase assignment, add a 2nd one
DeclarativeBase2 = declarative_base()

# uncomment the metadata2 line and assign it to DeclarativeBase2.metadata
metadata2 = DeclarativeBase2.metadata

# finally, modify the init_model method to allow both engines to be passed (see previous step)
# and assign the sessions and metadata to each engine
def init_model(engine1, engine2):
    """Call me before using any of the tables or classes in the model."""

#     DBSession.configure(bind=engine)
    DBSession.configure(bind=engine1)
    DBSession2.configure(bind=engine2)

    metadata.bind = engine1
    metadata2.bind = engine2

```

Tell Your Models Which Engine To Use

Now that the configuration has all been taken care of, you can instruct your models to inherit from either the first or second DeclarativeBase depending on which DB engine you want it to use.

For example, in `myapp/model/spam.py` (uses engine1):

```

from sqlalchemy import Table, ForeignKey, Column
from sqlalchemy.types import Integer, Unicode, Boolean
from myapp.model import DeclarativeBase

class Spam(DeclarativeBase):
    __tablename__ = 'spam'

    def __init__(self, id, variety):
        self.id = id
        self.variety = variety

    id = Column(Integer, autoincrement=True, primary_key=True)
    variety = Column(Unicode(50), nullable=False)

```

And then in `myapp/model/eggs.py` (uses `engine2`):

```
from sqlalchemy import Table, ForeignKey, Column
from sqlalchemy.types import Integer, Unicode, Boolean
from myapp.model import DeclarativeBase2

class Eggs(DeclarativeBase2):
    __tablename__ = 'eggs'

    def __init__(self, id, pkg_qty):
        self.id = id
        self.pkg_qty = pkg_qty

    id = Column(Integer, autoincrement=True, primary_key=True)
    pkg_qty = Column(Integer, default=12)
```

If you needed to use the `DBSession` here (or in your controllers), you would use `DBSession` for the 1st engine and `DBSession2` for the 2nd (see the previous and next sections).

Optional: Create And Populate Each Database In `Websetup.py`

If you want your `setup_app` method to populate each database with data, simply use the appropriate meta-data/`DBSession` objects as you would in a single-db setup.

In `myapp/websetup.py`:

```
def setup_app(command, conf, vars):
    """Place any commands to setup myapp here"""
    load_environment(conf.global_conf, conf.local_conf)
    # Load the models
    from myapp import model
    print "Creating tables for engine1"
    model.metadata.create_all()
    print "Creating tables for engine2"
    model.metadata2.create_all()

    # populate spam table
    spam = [model.Spam(1, u'Classic'), model.Spam(2, u'Golden Honey Grail')]
    # DBSession is bound to the spam table
    model.DBSession.add_all(spam)

    # populate eggs table
    eggs = [model.Eggs(1, 12), model.Eggs(2, 6)]
    # DBSession2 is bound to the eggs table
    model.DBSession2.add_all(eggs)

    model.DBSession.flush()
    model.DBSession2.flush()
    transaction.commit()
    print "Successfully setup"
```

Todo

Difficulty: Hard. At some point, we should also find a way to document how to handle [Horizontal and Vertical Partitioning](#) properly, and document that in here, too.

4.3.5 Using LDAP for user authentication and authorization

Status RoughDoc

Table of Contents

- Using LDAP for user authentication and authorization

This recipe shows how to configure TurboGears to use an LDAP directory for user authentication and authorization.

Requirements

We will use the `who_ldap` plugin for `repoze.who` for this purpose, since it supports both Python 2 and Python 3.

You can find the [documentation for who_ldap](#) on PyPi. From there, you can also install `who_ldap`, just run:

```
pip install who_ldap
```

You should also add this requirement to your project's `setup.py` file:

```
install_requires=[
    ...
    "who_ldap",
]
```

Note that `who_ldap` itself requires the `ldap3` package (formerly known as `python3-ldap`), which is a pure Python implementation of an LDAP v3 client. See the [documentation for ldap3](#) for details.

Configuration

Here is an example configuration that you can put into the `config/app_cfg.py` file of your project:

```
# Configure the base SQLAlchemy setup:
base_config.use_sqlalchemy = False

# Configure the authentication backend:

# YOU MUST CHANGE THIS VALUE IN PRODUCTION TO SECURE YOUR APP
base_config.sa_auth.cookie_secret = 'secret'

base_config.auth_backend = 'ldapauth'

from who_ldap import (LDAPSearchAuthenticatorPlugin,
                      LDAPAttributesPlugin, LDAPGroupsPlugin)

# Tell TurboGears how to connect to the LDAP directory

ldap_url = 'ldaps://ad.snake-oil-company.com'
ldap_base_dn = 'ou=users,dc=ad,dc=snake-oil-company,dc=com'
ldap_bind_dn = 'cn=bind,cn=users,dc=ad,dc=snake-oil-company,dc=com'
ldap_bind_pass = 'silverbullet'

# Authenticate users by searching in LDAP

ldap_auth = LDAPSearchAuthenticatorPlugin(
    url=ldap_url, base_dn=ldap_base_dn,
```

```
bind_dn=ldap_bind_dn, bind_pass=ldap_bind_pass,
returned_id='login',
# the LDAP attribute that holds the user name:
naming_attribute='sAMAccountName',
start_tls=True)

base_config.sa_auth.authenticators = [('ldapauth', ldap_auth)]

# Retrieve user metadata from LDAP

ldap_user_provider = LDAPAttributesPlugin(
    url=ldap_url, bind_dn=ldap_bind_dn, bind_pass=ldap_bind_pass,
    name='user',
    # map from LDAP attributes to TurboGears user attributes:
    attributes='givenName=first_name,sn=last_name,mail=email_address',
    flatten=True, start_tls=True)

# Retrieve user groups from LDAP

ldap_groups_provider = LDAPGroupsPlugin(
    url=ldap_url, base_dn=ldap_base_dn,
    bind_dn=ldap_bind_dn, bind_pass=ldap_bind_pass,
    filterstr='(&(objectClass=group)(member=%(dn)s))',
    name='groups',
    start_tls=True)

base_config.sa_auth.mdproviders = [
    ('ldapuser', ldap_user_provider),
    ('ldapgroups', ldap_groups_provider)]

from tg.configuration.auth import TGAAuthMetadata

class ApplicationAuthMetadata(TGAAuthMetadata):
    """Tell TurboGears how to retrieve the data for your user"""

    # map from LDAP group names to TurboGears group names
    group_map = {'operators': 'managers'}

    # set of permissions for all mapped groups
    permissions_for_groups = {'managers': {'manage'}}

    def __init__(self, sa_auth):
        self.sa_auth = sa_auth

    def get_user(self, identity, userid):
        user = identity.get('user')
        if user:
            name = '{first_name} {last_name}'.format(**user).strip()
            user.update(user_name=userid, display_name=name)
        return user

    def get_groups(self, identity, userid):
        get_group = self.group_map.get
        return [get_group(g, g) for g in identity.get('groups', [])]

    def get_permissions_for_group(self, group):
        return self.permissions_for_groups.get(group, set())
```

```

def get_permissions(self, identity, userid):
    permissions = set()
    get_permissions = self.get_permissions_for_group
    for group in self.get_groups(identity, userid):
        permissions |= get_permissions(group)
    return permissions

base_config.sa_auth.authmetadata = ApplicationAuthMetadata(
    base_config.sa_auth)

# Override this if you would like to provide a different who plugin for
# managing login and logout of your application:

base_config.sa_auth.form_plugin = None

# Page where you want users to be redirected to on login:

base_config.sa_auth.post_login_url = '/post_login'

# Page where you want users to be redirected to on logout:

base_config.sa_auth.post_logout_url = '/post_logout'

```

You will need to change the connection parameters to point to your user base in your LDAP directory and login with a bind user and password that is authorized to search over the directory.

You may also need to change some of the other parameters according to your requirements. The configuration for retrieving user metadata and user groups is optional if you want to use LDAP solely for authentication and not for authorization.

4.3.6 Autogenerated Forms with Sprox

This is a succinct explanation on how to use sprox's form rendering capabilities within TurboGears2. We will assume the reader is somewhat versed in TurboGears2's `tg.controllers.RestController`. Note that this is the same technology the Turbogears2 admin is based on, so this knowledge is valuable to understand how to configure the admin for your purposes.

Establishing the Model Definition

Let us first assume the following model for this demonstration.:

```

from sqlalchemy import Column, Integer, String, Date, Text, ForeignKey, Table
from sqlalchemy.orm import relation

from moviedemo.model import DeclarativeBase, metadata

movie_directors_table = Table('movie_directors', metadata,
                              Column('movie_id', Integer, ForeignKey('movies.movie_id'), primary_key=True),
                              Column('director_id', Integer, ForeignKey('directors.director_id'), primary_key=True))

class Genre(DeclarativeBase):
    __tablename__ = "genres"
    genre_id = Column(Integer, primary_key=True)
    name = Column(String(100))
    description = Column(String(200))

```

```
class Movie(DeclarativeBase):
    __tablename__ = "movies"
    movie_id = Column(Integer, primary_key=True)
    title = Column(String(100), nullable=False)
    description = Column(Text, nullable=True)
    genre_id = Column(Integer, ForeignKey('genres.genre_id'))
    genre = relation('Genre', backref='movies')
    release_date = Column(Date, nullable=True)

class Director(DeclarativeBase):
    __tablename__ = "directors"
    movie_id = Column(Integer, primary_key=True)
    title = Column(String(100), nullable=False)
    movies = relation(Movie, secondary_join=movie_directors_table, backref="directors")
```

The Basic Sprox Form

Here is how we create a basic form for adding a new Movie to the database:

```
class NewMovieForm(AddRecordForm):
    __model__ = Movie
new_movie_form = NewMovieForm(DBSession)
```

And our controller code would look something like this:

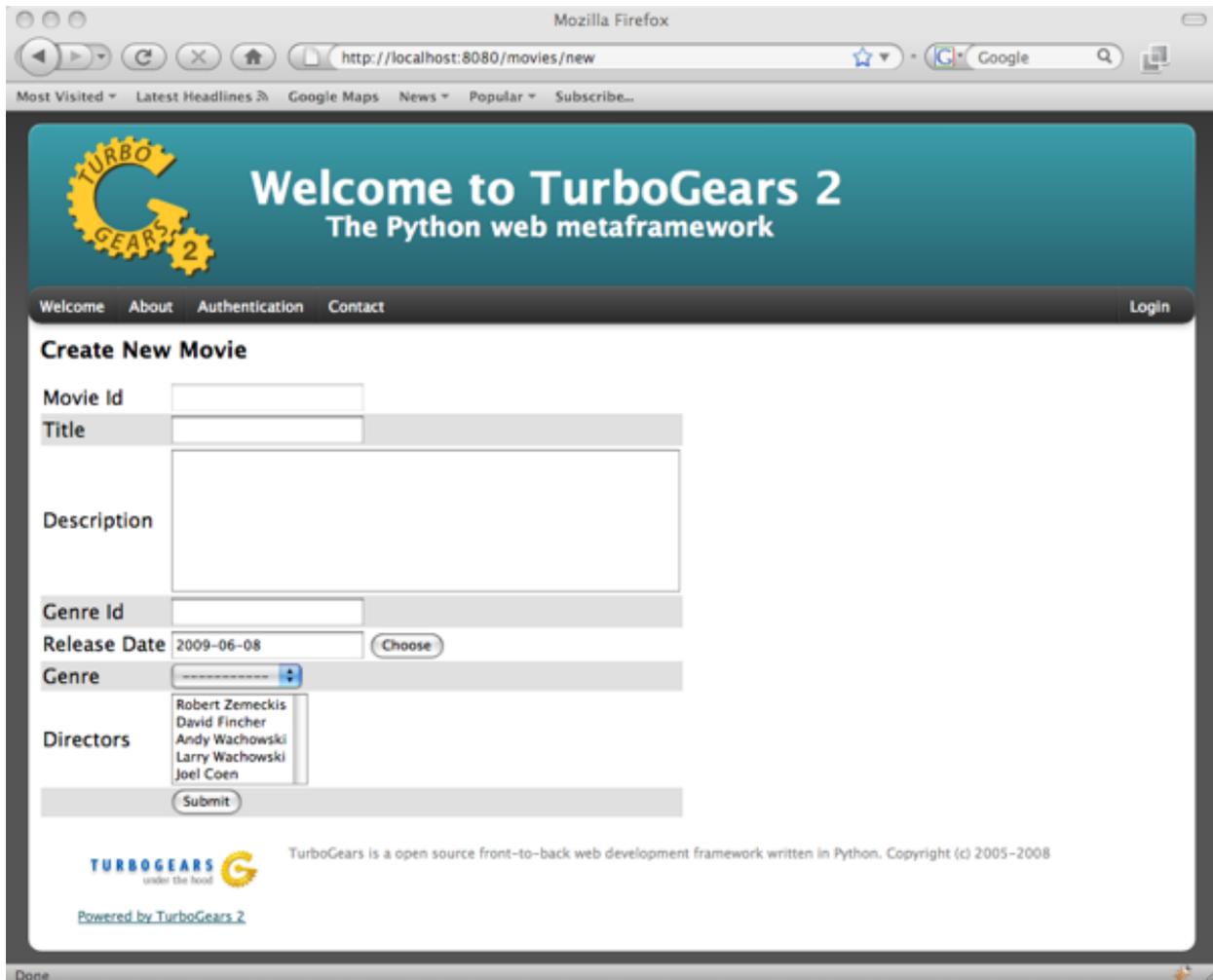
```
@expose('moviedemo.templates.sproxdemo.movies.new')
def new(self, **kw):
    tmpl_context.widget = new_movie_form
    return dict(value=kw)
```

You may have noticed that we are passing keywords into the method. This is so that the values previously typed by the user can be displayed on failed validation.

And finally, our template code:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:py="http://genshi.edgewall.org/"
      xmlns:xi="http://www.w3.org/2001/XInclude">
  <xi:include href="master.html" />
</html>
<head/>
<body>
  <div style="height:0px;" &nbsp;</div>
  <div>
    <div style="float:left width: 80%">
      <h2 style="margin-top:1px;">Create New Movie</h2>
      ${tmpl_context.widget (value=value)}
    </div>
  </div>
</body>
</html>
```

Which produces a form like this:

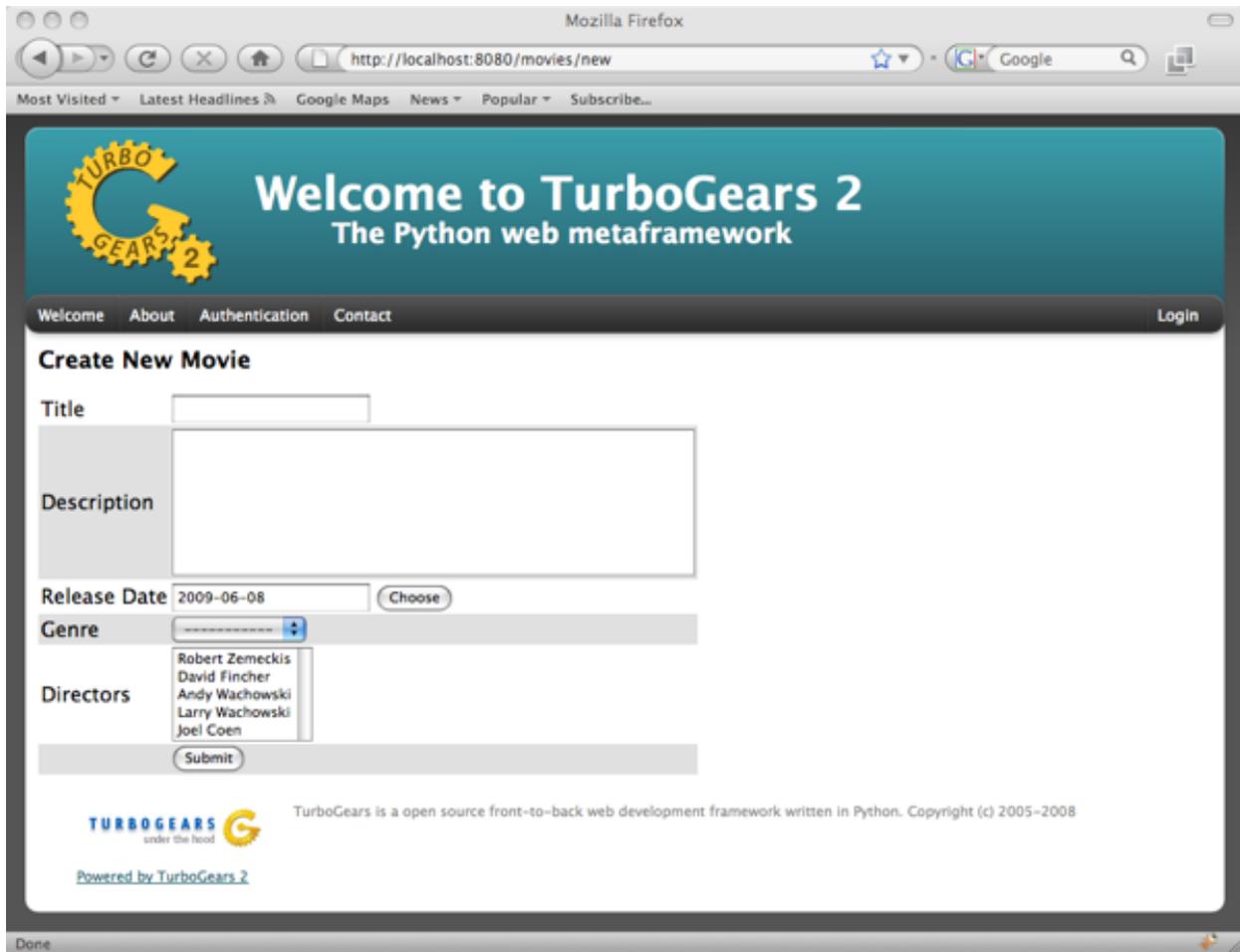


Omitting Fields

Now, we can use the `__omit_fields__` modifier to remove the “movie_id” and “genre_id” fields, as they will be of little use to our users. Our form code now becomes:

```
class NewMovieForm(AddRecordForm):
    __model__ = Movie
    __omit_fields__ = ['genre_id', 'movie_id']
```

The rendered form now looks like this:



Limiting fields

If you have more omitted fields than required fields, you might want to use the `__limit_fields__` operator to eliminate the fields you don't want. The same above form will be rendered with the following code:

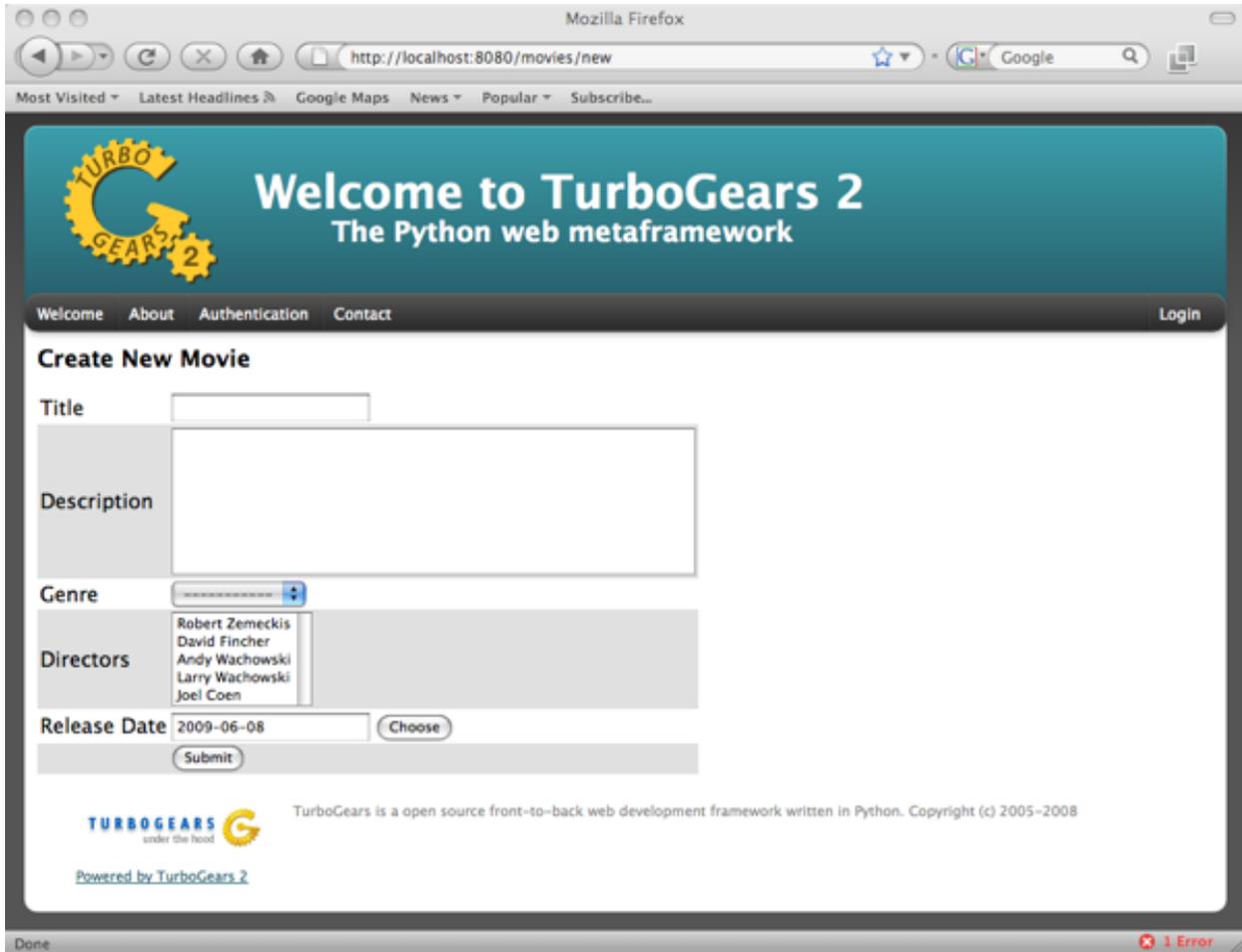
```
class NewMovieForm(AddRecordForm):
    __model__ = Movie[
        __limit_fields__ = ['title', 'description', 'release_date', 'genre', 'directors']
```

Field Ordering

If you want the fields displayed in a ordering different from that of the specified schema, you may use `field_ordering` to do so. Here is our form with the fields moved around a bit:

```
class NewMovieForm(AddRecordForm):
    __model__ = Movie
    __omit_fields__ = ['movie_id', 'genre_id']
    __field_order__ = ['title', 'description', 'genre', 'directors']
```

Notice how the `release_date` field that was not specified was still appended to the end of the form.

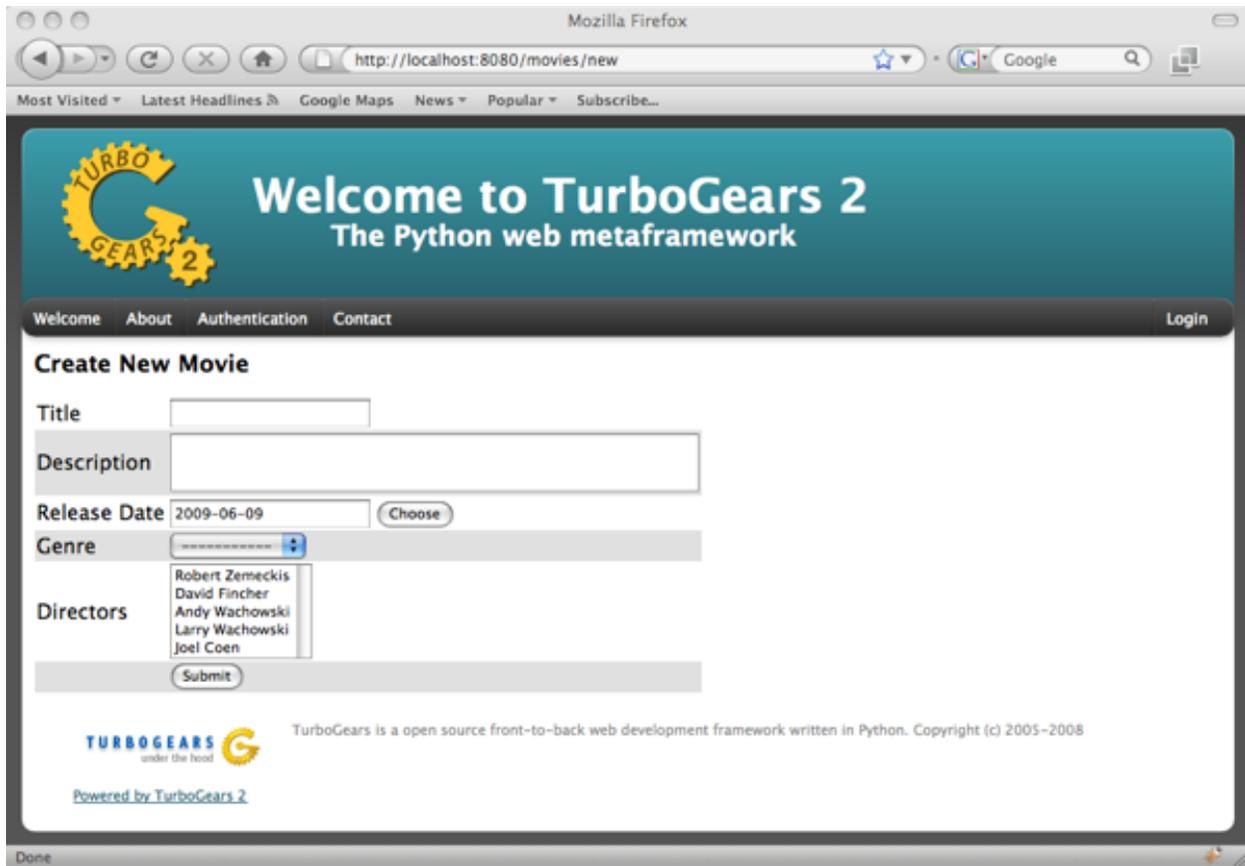


Overriding Field Attributes

Sometimes we will want to modify some of the HTML attributes associated with a field. This is as easy as passing a `__field_attrs__` modifier to our form definition. Here is how we could modify the description to have only 2 rows:

```
class NewMovieForm (AddRecordForm) :
    __model__ = Movie
    __omit_fields__ = ['movie_id', 'genre_id']
    __field_attrs__ = {'description': {'rows': '2'}}
```

Here is the resultant form:

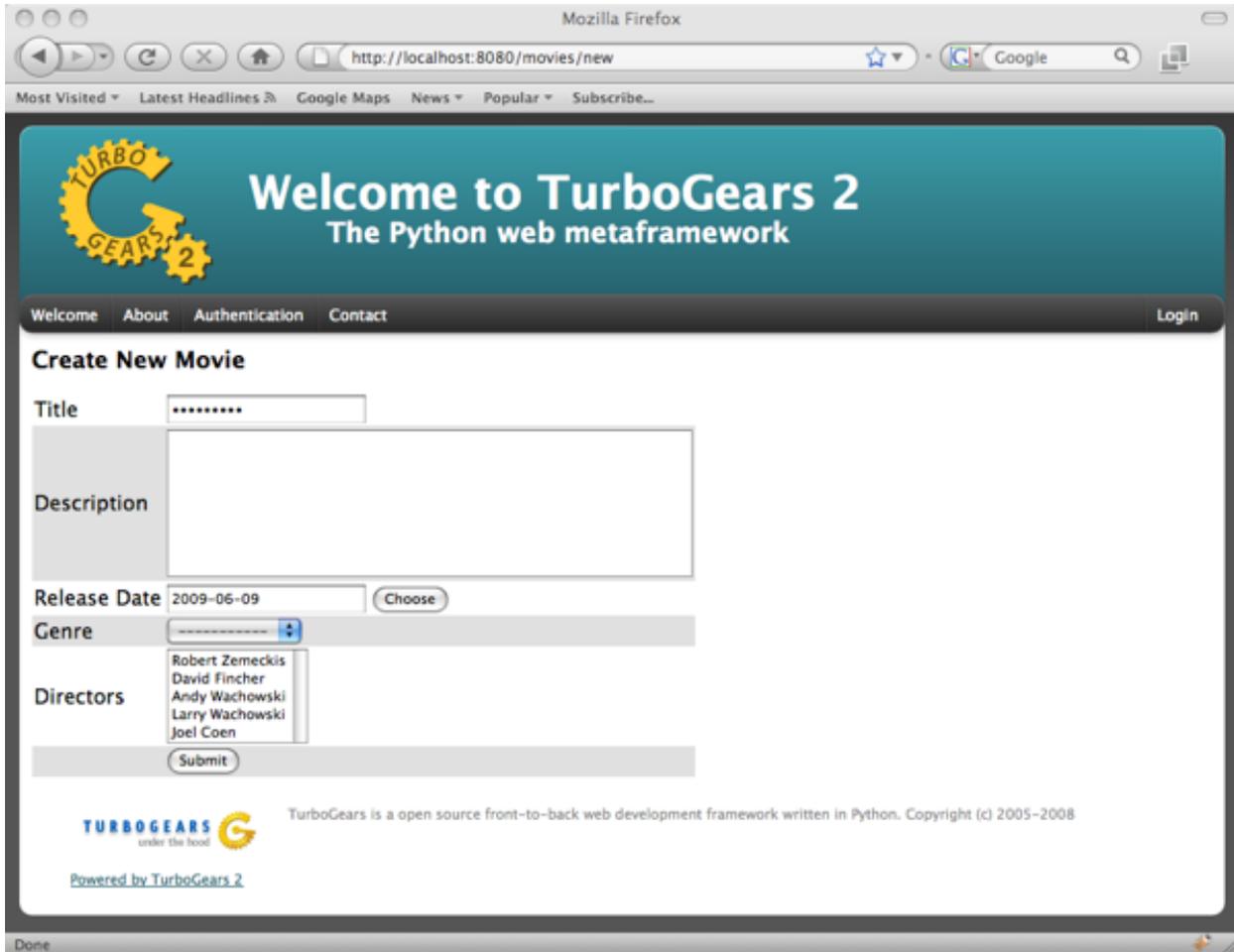


Overriding a Form Field

Sometimes you want to override a field all together. Sprock allows you to do this by providing an attribute to your form class declaratively. Simply instantiate your field within the widget and it will override the widget used for that field. Let's change the movie title to a password field just for fun.:

```
from tw.forms.fields import PasswordField

class NewMovieForm(AddRecordForm):
    __model__ = Movie
    __omit_fields__ = ['movie_id', 'genre_id']
    __field_attrs__ = {'description':{'rows':'2'}}
    title = PasswordField
```



You can see now that the title is “starred” out. Note that you may also send an “instance” of a widget for a field, but you must pass in the fieldname to the widget. This is a limitation of ToscaWidgets. (You may not change the “id” of a widget after it has been created.):

```
title = PasswordField('title')
```

Field Widget Args

Sometimes you want to provide sprox with a class for a field, and have sprox set the arguments to a widget, but you either want to provide an additional argument, or override one of the arguments that sprox chooses. For this, pass a dictionary into the `__field_widget_args__` parameter with the key being the field you would like to pass the arg into, and the value a dictionary of args to set for that field. Here is an example of how to set the rows and columns for the description field of a form.:

```
class NewMovieForm(AddRecordForm):
    __model__ = Movie
    __field_widget_args__ = {'description': {'rows': 30, 'cols': 30}}
```

Custom Dropdown Field Names

Sometimes you want to display a field to the user for the dropdown that has not been selected by sprox. This is easy to override. Simply pass the field names for the select boxes you want to display into the `__dropdown_field_names__`

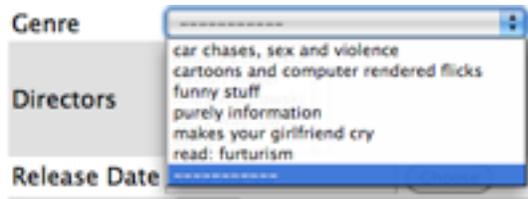
modifier:

```
class NewMovieForm(AddRecordForm):
    __model__ = Movie
    __omit_fields__ = ['movie_id', 'genre_id']
    __field_order__ = ['title', 'description', 'genre', 'directors']
    __dropdown_field_names__ = ['description', 'name']
```

If you want to be more specific about which fields should display which field, you can pass a dictionary into the `__dropdown_field_names__` modifier:

```
class NewMovieForm(AddRecordForm):
    __model__ = Movie
    __omit_fields__ = ['movie_id', 'genre_id']
    __field_order__ = ['title', 'description', 'genre', 'directors']
    __dropdown_field_names__ = {'genre': 'description', 'directors': 'name'}
```

Either will produce a new dropdown like this:



Creating Custom Dropdown Data

Sometimes providing a fieldname alone is not enough of a customization to inform your users into what they should be selecting. For this example, we will provide both name and description for the Genre field. This requires us to override the genre widget with one of our choosing. We will extend the existing sprox dropdown widget, modifying the `update_params` method to inject both name and description into the dropdown. This requires some knowledge of ToscaWidgets in general, but this recipe will work for the majority of developers looking to modify their dropdowns in a custom manner.

First, we extend the Sprox SingleSelect Field as follows:

```
from sprox.widgets import PropertySingleSelectField

class GenreField(PropertySingleSelectField):
    def __my_update_params(self, d, nullable=False):
        genres = DBSession.query(Genre).all()
        options = [(genre.genre_id, '%s (%s)'% (genre.name, genre.description))
                   for genre in genres]
        d['options'] = options
        return d
```

Then we include our new widget in the definition of the our movie form:

```
class NewMovieForm(AddRecordForm):
    __model__ = Movie
    __omit_fields__ = ['movie_id', 'genre_id']
    __field_order__ = ['title', 'description', 'genre', 'directors']
    __dropdown_field_names__ = {'genre': 'description', 'directors': 'name'}
    genre = GenreField
```

Here is the resulting dropdown:

Genre	action (car chases, sex and violence)
Directors	action (car chases, sex and violence) animation (cartoons and computer rendered flicks) comedy (funny stuff) documentary (purely information) drama (makes your girlfriend cry) sci-fi (read: futurism)
Release Date	2009-08-08 (Choose)

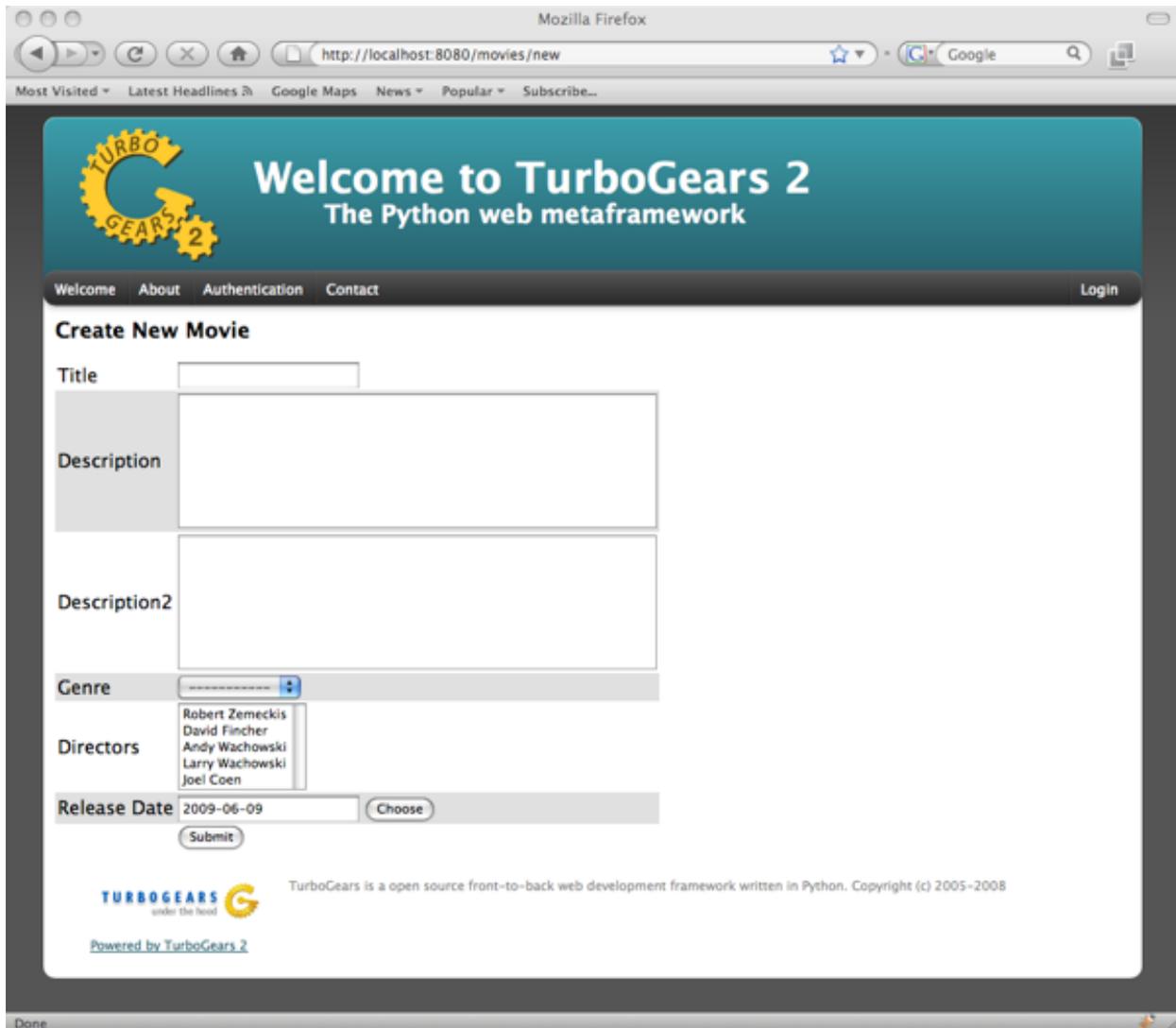
Adding a New Field

There may come a time when you want to add a field to your view which is not part of your database model. The classic case for this is password validation, where you want to provide a second entry field to ensure the user has provided a correct password, but you do not want/need that data to be stored in the database. Here is how we would go about adding a second description field to our widget.:

```
from tw.forms.fields import TextArea

class NewMovieForm (AddRecordForm) :
    __model__ = Movie
    __omit_fields__ = ['movie_id', 'genre_id']
    __field_order__ = ['title', 'description', 'description2', 'genre', 'directors']
    description2 = TextArea('description2')
```

For additional widgets, you must provide an instance of the widget since sprock will not have enough information about the schema of the widget in order to populate it correctly. Here's what our form now looks like:



Validation

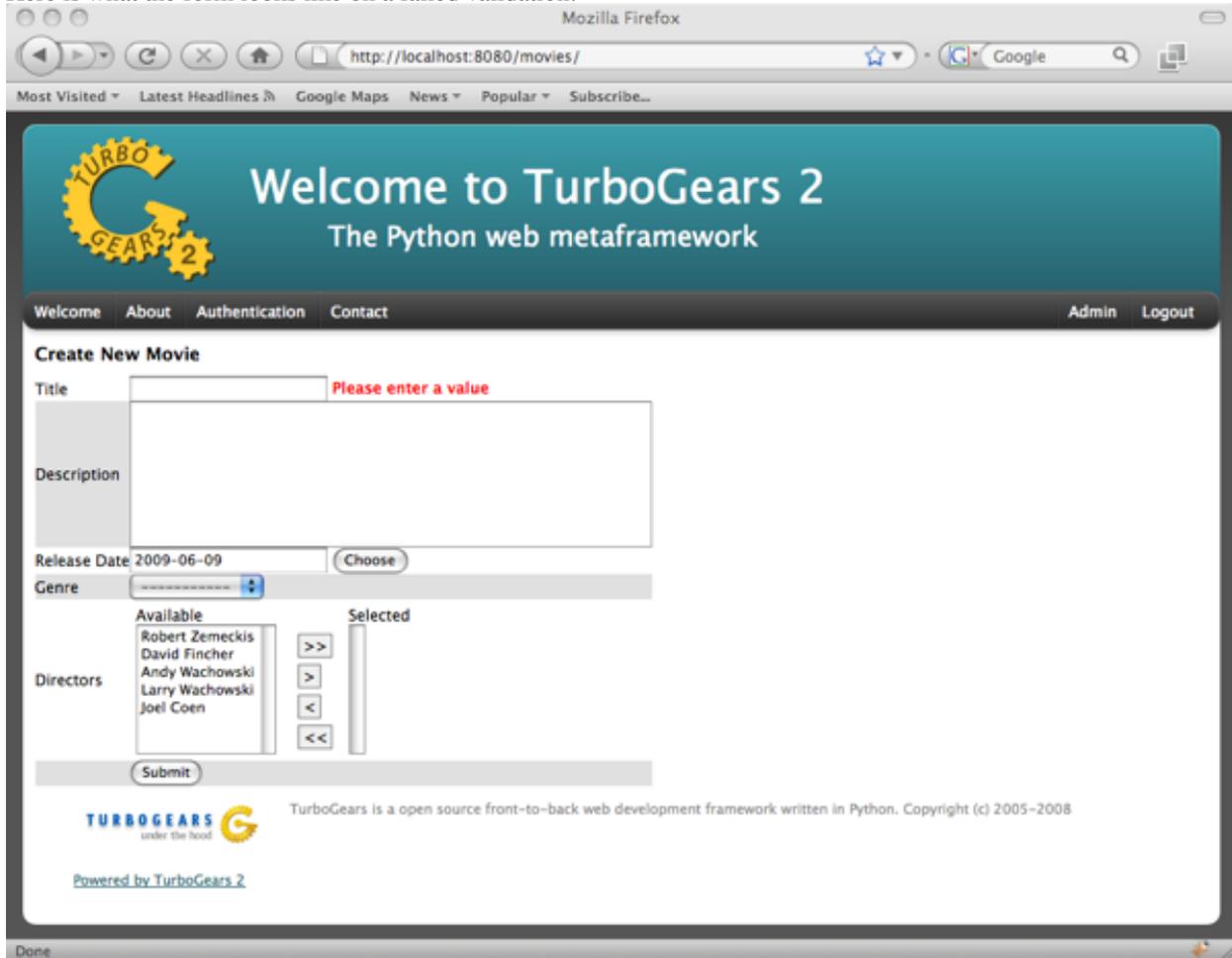
Turbogears2 has some great tools for validation that work well with sprox. In order to validate our form, we must first give the form a place to POST to, with a new method in our controller that looks like:

```
@validate(new_movie_form, error_handler=new)
@expose()
def post(self, **kw):
    del kw['sprox_id']
    kw['genre'] = DBSession.query(Genre).get(kw['genre'])
    kw['directors'] = [DBSession.query(Director).get(id) for id in kw['directors']]
    kw['release_date'] = datetime.strptime(kw['release_date'], "%Y-%m-%d")
    movie = Movie(**kw)
    DBSession.add(movie)
    flash('your movie was successfully added')
    redirect('/movies/')
```

A couple of things about this. First, we must remove the sprox_id from the keywords because they conflict with the Movie definition. This variable may go away in future versions. genre and directors both need to be converted into

their related objects before they are applied to the object, and the `release_date` needs to be formatted as a datetime object if you are using sqlite.

Here is what the form looks like on a failed validation:

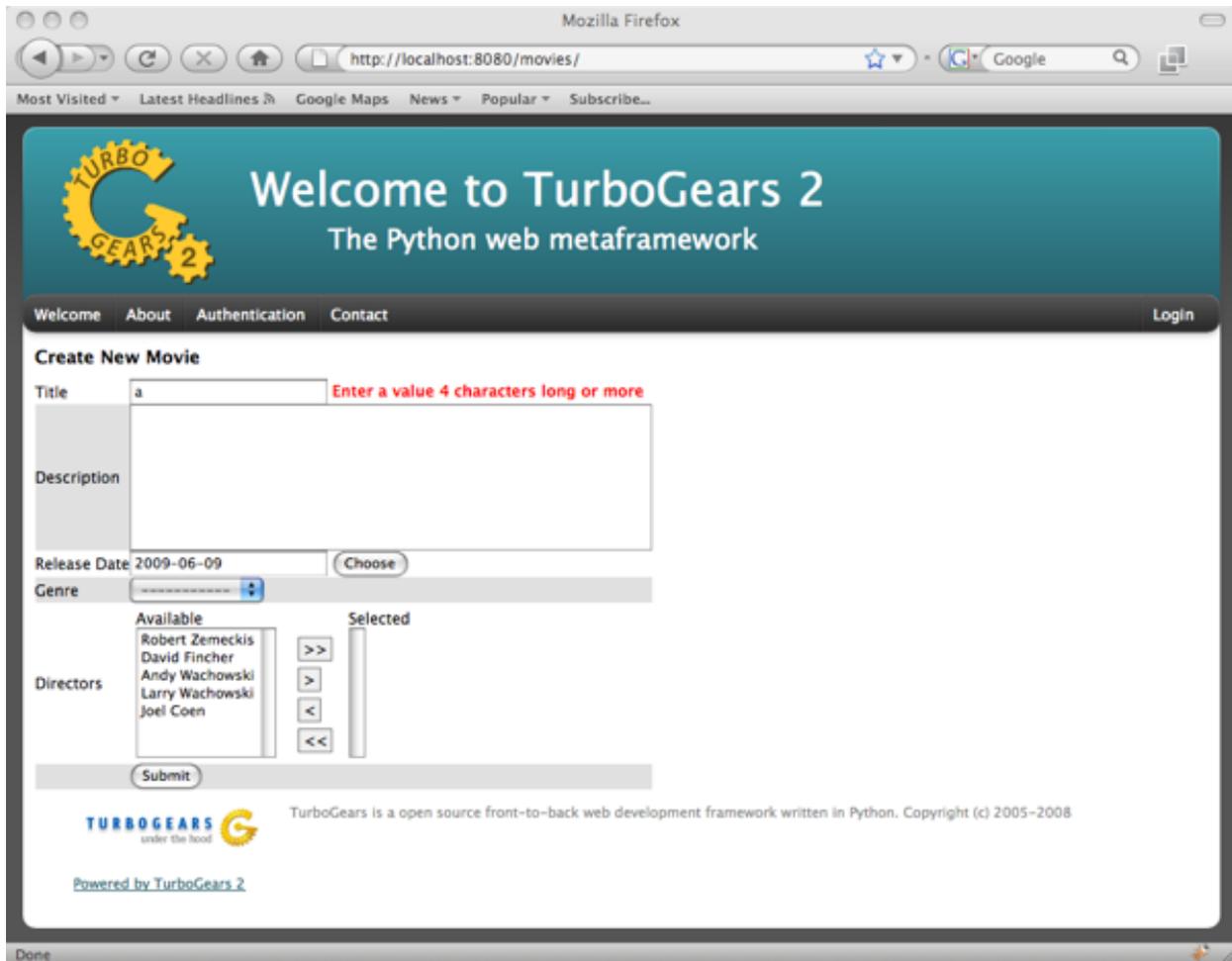


Overriding a Validator

Often times you will want to provide your own custom field validator. The best way to do this is to add the validator declaratively to your Form Definition:

```
from formencode.validators import String
class NewMovieForm(AddRecordForm):
    __model__ = Movie
    __omit_fields__ = ['movie_id', 'genre_id']
    title = String(min=4)
```

The resulting validation message looks like this:



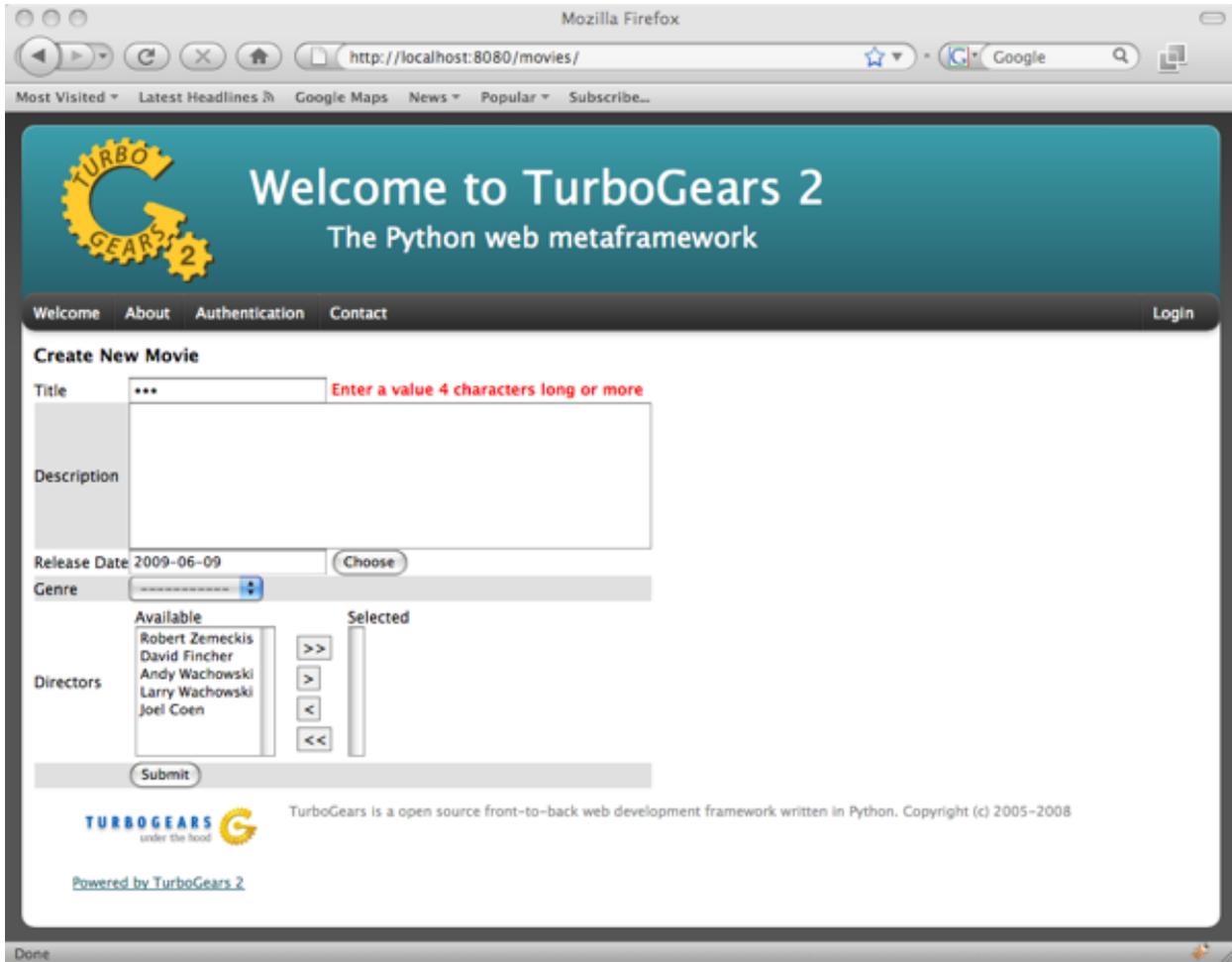
Overriding both Field and Validator

Ah, you may have realized that sometimes you must override both widget and validator. Sprox handles this too, by providing a `:class:sprox.formbase.Field` class that you can use to wrap your widget and validator together.:

```
from formencode.validators import String
from sprox.formbase import Field
from tw.forms.fields import PasswordField

class NewMovieForm(AddRecordForm):
    __model__ = Movie
    __omit_fields__ = ['movie_id', 'genre_id']
    title = Field(PasswordField, String(min=4))
```

Again, the field class does not care if you pass instances or class of the widget.

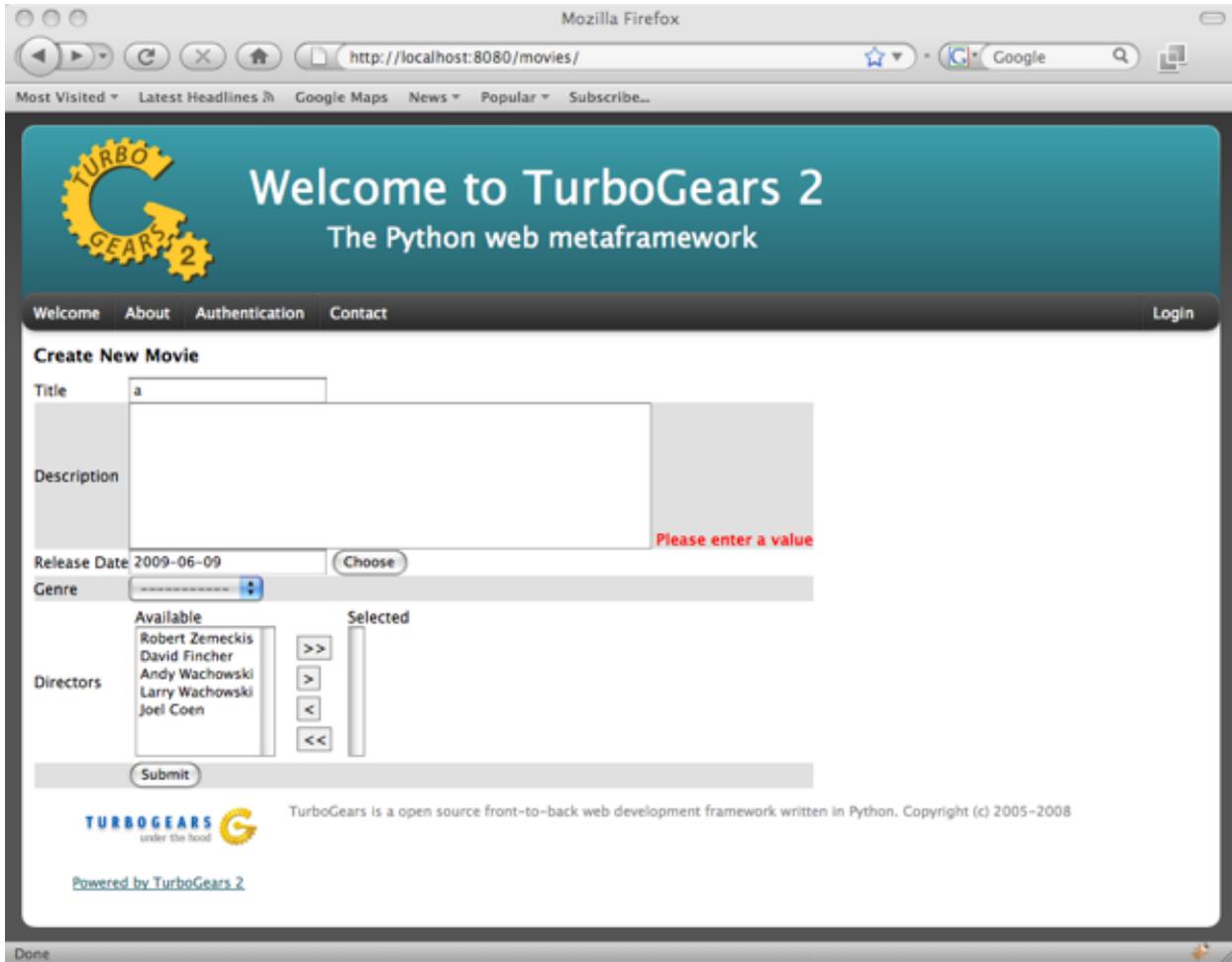


Required Fields

You can tell sprock to make a field required even if it is nullable in the database by passing the fieldname into a list of the `__require_fields__` modifier.:

```
class NewMovieForm (AddRecordForm) :
    __model__ = Movie
    __omit_fields__ = ['movie_id', 'genre_id']
    __require_fields__ = ['description']
```

And the form now sports a validation error:



Form Validation

You can validate at the form level as well. This is particularly interesting if you need to compare two fields. See *TurboGears Validation*.

Conclusion

`sprox.formbase.FormBase` class provides a flexible mechanism for creating customized forms. It provides sensible widgets and validators based on your schema, but can be overridden for your own needs. `FormBase` provides declarative addition of fields, ways to limit and omit fields to a set that is appropriate for your application. `Sprox` provides automated drop-down boxes, as well as providing a way to override those widgets for your purposes.

4.4 Contributing to TurboGears

4.4.1 Preparing Your Development Environment

Installing and Using Git

Please refer to the [Git](#) site for directions on how to install a current version of Git on your system. Note that we do not recommend using a version less than 1.5 for working with Git. Versions earlier than that seemed overly complex to use.

The best way to use a version control system (and especially a distributed version control system like [Git](#)) is a subject that could span several books.

Instead of going through all of the detail of the many ways to use [Git](#), we refer you to the [Git documentation](#) site for a long list of tutorials, online documentation, and books (both paper and ebook) for you to read that will teach you the many options you can use with [Git](#).

Create A virtualenv

As stated in [virtualenv Notes](#), a virtualenv is extremely recommended for development work. Make a new blank virtualenv and activate it. This way, you will have all of your work isolated, preventing conflicts with anything else on which you might be working.

Do not do any easy_install's of any sort. We will cover that in the next step.

Installing TurboGears2

On the [TurboGears2](#) project pages, there are a bunch of repositories related to the turbogears project. The most important are:

TG2.x Core This is the actual core of TurboGears2. Unless you are working on modifying a template or one of the [Gearbox](#) based tools, or even the documentation, this is the repository you want.

TG2.x Devtools This repository is the add-on tools. It gets updated when you wish to make a change to help an application developer (as opposed to an application installer). It contains all the stock TurboGears2 templates, and references the [Gearbox](#) toolchain to provide an HTTP server, along with other tools.

TG2.x Docs This repository contains two versions of the documentation. The first version (located in the docs directory) is the older docs, and is gradually being phased out. The newer version (located under the book directory) contains this file (and others) and is gradually being brought on par with the old, and will eventually replace the older version entirely.

The best way to prepare your development environment is to take the following steps:

1. Clone the first three repositories ([TG2.x Core](#), [TG2.x Devtools](#), and [TG2.x Docs](#)).
2. Enter the top level directory for your TG2.x Core clone, and run `python setup.py tgdevelop` and `python setup.py tgdeps`
3. Enter the top level directory for your TG2.x Devtools clone, and run `python setup.py tgdevelop` and `python setup.py tgdeps`
4. Enter the `book` directory for your TG2.x Docs clone, and run `python setup.py tgdevelop` and `python setup.py tgdeps`

After you've done all this, you have a working copy of the code sitting in your system. You can explore the code and begin working through any of the [tickets](#) you wish, or even on your own new features that have not yet been submitted.

Note that, for all repositories, work is to be done off of the `development` branch. Either work directly on that branch, or do the work on a branch made from the `development` branch. The `master` branch is reserved for released code.

When working on your feature or ticket, make certain to add the test cases. Without them, the code will not be accepted.

4.4.2 Testing Your Changes

After doing your development work, before sending to the main repositories (or sending in a pull request), you must make sure that your code does not break anything.

This process is actually rather painless. The first time you do it, you need to run `python setup.py nosetests` from the top of your working tree. This will download any missing packages that are required for testing, and then run the tests.

After that first time, you may use `nosetests`, and all of the tests will be run.

In either case, you will be told about any failures. If you have any, either fix the code or (if the test case is wrong) fix the test. Then re-run the tests.

If you are interested, you can also see the current status of the tests, and how much of the core code is actually being tested. Run this command:

```
$ nosetests --with-coverage --cover-erase --cover-package=tg
```

You will now see which lines are being tested, which ones are not, and have a thorough report on the status of the testing coverage.

4.4.3 Preparing a Release of TurboGears

Prerequisites

1. You have a working knowledge of how to use a `virtualenv`.
2. You have shell access to the `turbogears` site.
3. You know how to run the `nosetests` on your local git clones of TurboGears.
4. You have to have a working knowledge of `git`, and how to do merging and branching.
5. You need permission to update the TurboGears2 and `tg.devtools` packages on `PyPI`

With those prerequisites in mind, this document will not cover all the command lines that you could run. Instead, we will specify steps as “activate your `virtualenv`” or “run `nosetests`” or the like.

Summary of Steps

Preparing a release of TurboGears is going to take some time and effort. It is unlikely to be completed in a mere day or two, so please plan on taking some time to work through it all.

The steps for the release, in summary, are as follows:

1. Repository Branching
2. Finalizing Changes On ‘next’ Branch
3. Preparing Changelog And Release Announcement

4. Preparing Packages And The Documentation
5. Uploading The Documentation
6. Pushing to PyPI
7. Publishing Release Announcement And Closing Milestones
8. Final Cleanup

Below, we discuss each of these steps in detail.

Repository Branching

We have three main repositories: `TG2`, `TG2Devtools`, and `TG2Docs`. Each of them functions in a similar fashion for this process:

1. Clone the repository
2. Checkout the “development” branch and update it
3. Checkout the “next” branch and merge development into it

Once the “next” branch is made, this is where you will do all work until the release is done.

Finalizing Changes On ‘next’ Branch

After all the changes that you’ve made so far, the final changes are simply to get the new version numbers into the distributed files.

- In `TG2`:
 - Update `tg/release.py` to have the new version number.
- In `TG2Devtools`:
 - Update `setup.py`:
 - * Update the version number
 - * Update the install requirements so that it requires TurboGears2 \geq the new version number
 - Update `devtools/templates/turbogears/setup.py_tmpl`:
 - * Update the install requirements so that it requires TurboGears2 \geq the new version number
 - Update `CHANGES.txt`:
 - * Add new release with changelog generate with: `git log --no-merges --format="* %s" LAS_RELEASE_TAG..HEAD` (`LAST_RELEASE_TAG` is the tag of the previous release, like `tg2.3.2`). Please take only meaningful changes.
- In `TG2Docs`:
 - Update `requirements.txt`:
 - * Change TurboGears dependency line from `@development` to current release (EX: `@tg2.3.2`).
 - Update `docs/config.py`:
 - * Update the version number

Commit all of these changes, but do not push them public, not yet.

Preparing Changelog And Release Announcement

For each of the three repositories, you will need to review the commit logs since the last release. Gather up the summaries for each one, and prepare a new file. Use the standard [GNU Changelog](#) format. However, instead of recording individual file changes, record only the summaries. We don't need the file changes since Git records those changes for us.

Review the [GitHub](#) tickets for this milestone, and record any tickets that were closed for this repository but were not referenced in the summaries you've already recorded.

The changelog files you've made will be the commit message for the tags you are about to make.

In addition, prepare a release announcement. Anything I can say here sounds condescending. You should prepare it, though, so that as soon as you reach the "Publish" step, it's all done in a few minutes.

Preparing Packages And The Documentation

First, merge the branch "next" onto the branch "master". Then, tag the master branch with the new version number, and use the changelog you've generated as the commit message. The tag should be an annotated tag (i.e.: `git tag -a`).

Do this for each of the three repositories.

For the documentation, go into the appropriate directory, and type `make html` (either the docs or the book, whichever is needed to be uploaded).

Uploading The Documentation

When you run `make html`, it will create a directory "`_build/html`". Upload the contents of that directory and replace the current directory with it. For instance, if you used `rsync` to upload to your user account on the server, and fixed the permissions so that the website user could read the files, you could then do `rsync -avP --delete /path/to/new/docs /path/to/web/docs/directory` and have everything properly uploaded/visible to the users.

Do not forget the book! Enter the `tg2docs/book` folder, and run `make html`. This will produce the necessary `html` files for the book. Upload the contents of the `book/_build/html` directory to the webserver. Use similar commands as were used for copying the older `html` docs to complete the process.

Making The Source Distribution For The New Eggbasket

At this point, everything is prepared, with one exception: The source distributions for TurboGears2 and `tg.devtools` must be placed in the `eggbasket`. Enter your local repository directory for both `TG2.x Core` and `TG2.x DevTools` and run `python setup.py sdist`. In both of them, you will produce a directory named `dist` with a `.tar.gz` file for the new version. Copy these files to your `${HOME}/eggbasket`, then go to `${HOME}/eggbasket` and run `makeindex *`.

Using the steps in [Testing Jenkins With The Upgraded Packages And Code](#), upload the updated (and finalized) `eggbasket` to the `turbogears.org` web server.

Making The New Eggbasket The Current On Turbogears.org

Log in to the [turbogears](#) website. Go into the directory where you stored the "next" directory, and rename "next" to the version you are releasing. Remove the "current" link, and then do a symbolic link from the version being released to "current", like so: `ln -s 2.1.1 current`

Pushing to PyPI

For all three repositories, do `python setup.py upload`.

Publishing Release Announcement And Closing Milestones

Publish your release announcement to the places of your choice. We recommend your blog(s) and twitter. In addition, update the [turbogears](#) “Current Status” page to reflect the new release.

Final Cleanup

For each of the three repositories, merge the “master” branch to the “development” branch.

You’re done. Sit back and enjoy having accomplished a release.

TurboGears Reference

5.1 Configuration Options

This section reports a reference of various configuration options available inside TurboGears `app_cfg` or `.ini` file. As they are automatically extracted from the source code this list might be incomplete until the whole configuration process is updated to automatically declare the expected options.

For example if you want to enable isodates in your application JSON encoder you might want to add to your `app_cfg.base_config` the following:

```
base_config['json.isodates'] = True
```

5.1.1 JSON Encoding

`JSONEncoder.configure` (*isodates=False, custom_encoders=None, **kwargs*)

JSON encoder can be configured through `AppConfig` (`app_cfg.base_config`) using the following options:

- `json.isodates` -> encode dates using ISO8601 format
- `json.custom_encoders` -> List of tuples (`type, encode_func`) to register custom encoders for specific types.

5.1.2 Flash Messages

`TGFlash.configure` (*cookie_name='webflash', default_status='ok', template=<string.Template object at 0x7f0d1dc56510>, js_call='webflash.render()', js_template=<string.Template object at 0x7f0d1dc56550>*)

Flash messages can be configured through `AppConfig` (`app_cfg.base_config`) using the following options:

- `flash.cookie_name` -> Name of the cookie used to store flash messages
- `flash.default_status` -> Default message status if not specified (ok by default)
- `flash.template` -> `string.Template` instance used as the flash template when rendered from server side, will receive `$container_id`, `$message` and `$status` variables.
- `flash.js_call` -> javascript code which will be run when displaying the flash from javascript. Default is `webflash.render()`, you can use `webflash.payload()` to retrieve the message and show it with your favourite library.

- `flash.js_template` -> `string.Template` instance used to replace full javascript support for flash messages. When rendering flash message for javascript usage the following code will be used instead of providing the standard `webflash` object. If you replace `js_template` you must also ensure cookie parsing and delete it for already displayed messages. The template will receive: `$container_id`, `$cookie_name`, `$js_call` variables.

5.1.3 Sessions

class `tg.appwrappers.session.SessionApplicationWrapper` (*handler, config*)
Provides the Session Support

The Session Application Wrapper will make a lazy session instance available every request under the `environ['beaker.session']` key and inside TurboGears context as `session`.

Supported options which can be provided by config are:

- `session.enabled`: Whenever sessions are enabled or not.
- Beaker Options prefixed with `session.`, see <https://beaker.readthedocs.org/en/latest/configuration.html#session-options>

5.1.4 Caching

class `tg.appwrappers.caching.CacheApplicationWrapper` (*handler, config*)
Provides Caching Support.

The Cache Application Wrapper will make a `CacheManager` instance available every request under the `environ['beaker.cache']` key and inside the TurboGears request context as `cache`.

Supported options which can be provided by config are:

- `cache.enabled`: Whenever caching is enabled or not.
- Beaker Options prefixed with `cache.`, see <https://beaker.readthedocs.org/en/latest/configuration.html#cache-options>

5.1.5 Internationalization

class `tg.appwrappers.i18n.I18NApplicationWrapper` (*handler, config*)
Provides Language detection from request and session.

The session language(s) take priority over the request languages.

Supported options which can be provided by config are:

- `i18n.enabled`: Whenever language detection is enabled or not.
- `i18n.lang`: Fallback language for the application, works both when language detection is enabled or disabled. If this is set and language detection is disabled, the application will consider that all gettext wrapped strings must be translated to this language.
- `i18n.lang_session_key`: Session key from which to read the saved language (`tg_lang` by default).
- `i18n.no_session_touch`: Avoid causing a session save when reading it to retrieve the favourite user language. This is `False` by default, setting it to `False` causes TurboGears to save and update the session for each request.

5.1.6 Transaction Manager

class `tg.appwrappers.transaction_manager.TransactionApplicationWrapper` (*handler, config*)

Wraps the whole application in zope.transaction transaction manager and rollbacks transaction in case of crashes.

Supported options which can be provided by config are:

- `tm.enabled`: Whenever the transaction manager is enabled or not.
- `tm.attempts`: Number of times the transaction should be retried if it fails (no retry by default)
- `tm.commit_veto`: A function that will be called for every transaction to check if it should abort transaction or let it go. Function signature should be: `function(environ, status_code, headers) -> bool`.

5.1.7 Custom Error Pages

class `tg.appwrappers.errorpage.ErrorPageApplicationWrapper` (*handler, config*)

Given an Application it intercepts the response code and shows a custom page.

Supported options are:

- `errorpage.enabled`: Whenever the custom error page is enabled or not.
- `errorpage.status_codes`: List of HTTP errors that should be trapped. By default 403, 404, 500.
- `errorpage.handle_exceptions`: Whenever exceptions should be trapped and treated as a 500 error or not. By default this is True when `debug=false`.
- `errorpage.path`: Path of the controller should be displayed in case of errors. By default `/error/document`.

5.1.8 Ming Session Manager

class `tg.appwrappers.mingflush.MingApplicationWrapper` (*handler, config*)

Automatically flushes the Ming ODMSession.

In case an exception raised during execution it won't flush the session and it will instead close it throwing away any change.

Supported options which can be provided by config are:

- `tm.autoflush`: Whenever to flush session at end of request if no exceptions happened.

5.2 Classes and Functions

This page provides a quick access reference to the classes and functions provided by TurboGears

5.2.1 Decorators

Decorators use by the TurboGears controllers.

Not all of these decorators are traditional wrappers. They are much simplified from the TurboGears 1 decorators, because all they do is register attributes on the functions they wrap, and then the DecoratedController provides the hooks needed to support these decorators.

class `tg.decorators.Decoration` (*controller*)

Simple class to support 'simple registration' type decorators

lookup_template_engine (*tgl*)

Return the template engine data.

Provides a convenience method to get the proper engine, content_type, template, and exclude_names for a particular tg_format (which is pulled off of the request headers).

register_custom_template_engine (*custom_format, content_type, engine, template, exclude_names, render_params*)

Registers a custom engine on the controller.

Multiple engines can be registered, but only one engine per custom_format.

The engine is registered when @expose is used with the custom_format parameter and controllers render using this engine when the use_custom_format() function is called with the corresponding custom_format.

exclude_names keeps track of a list of keys which will be removed from the controller's dictionary before it is loaded into the template. This allows you to exclude some information from JSONification, and other 'automatic' engines which don't require a template.

render_params registers extra parameters which will be sent to the rendering method. This allows you to influence things like the rendering method or the injected doctype.

register_template_engine (*content_type, engine, template, exclude_names, render_params*)

Registers an engine on the controller.

Multiple engines can be registered, but only one engine per content_type. If no content type is specified the engine is registered at / which is the default, and will be used whenever no content type is specified.

exclude_names keeps track of a list of keys which will be removed from the controller's dictionary before it is loaded into the template. This allows you to exclude some information from JSONification, and other 'automatic' engines which don't require a template.

render_params registers extra parameters which will be sent to the rendering method. This allows you to influence things like the rendering method or the injected doctype.

class `tg.decorators.after_render` (*hook_func*)

A list of callables to be run after the template is rendered.

Will be run before it is returned up the WSGI stack.

class `tg.decorators.before_call` (*hook_func*)

A list of callables to be run before the controller method is called.

class `tg.decorators.before_render` (*hook_func*)

A list of callables to be run before the template is rendered.

class `tg.decorators.before_validate` (*hook_func*)

A list of callables to be run before validation is performed.

class `tg.decorators.cached` (*key=<class 'tg.support.NoDefault'>, expire='never', type=None, query_args=None, cache_headers=('content-type', 'content-length'), invalidate_on_startup=False, cache_response=True, **b_kwargs*)

Decorator to cache the controller, if you also want to cache template remember to return tg_cache option from the controller.

The following parameters are accepted:

key - Specifies the controller parameters used to generate the cache key. NoDefault - Uses function name and all request parameters as the key (default)

None - No variable key, uses only function name as key

string - Use function name and only “key” parameter

list - Use function name and all parameters listed

expire Time in seconds before cache expires, or the string “never”. Defaults to “never”

type Type of cache to use: dbm, memory, file, memcached, or None for Beaker’s default

cache_headers A tuple of header names indicating response headers that will also be cached.

invalidate_on_startup If True, the cache will be invalidated each time the application starts or is restarted.

cache_response Determines whether the response at the time the cache is used should be cached or not, defaults to True.

Note: When `cache_response` is set to False, the `cache_headers` argument is ignored as none of the response is cached.

class `tg.decorators.decode_params` (*format='json'*)

Decorator that enables parsing parameters from request body.

By default the arguments are parsed in **JSON** format (which is currently the only supported format).

class `tg.decorators.expose` (*template='', content_type=None, exclude_names=None, custom_format=None, render_params=None, inherit=False*)

Register attributes on the decorated function.

Parameters

template Assign a template, you could use the syntax ‘genshi:template’ to use different templates. The default template engine is genshi.

content_type Assign content type. The default content type is ‘text/html’.

exclude_names Assign exclude names

custom_format Registers as a custom format which can later be activated calling `use_custom_format`

render_params Assign parameters that shall be passed to the rendering method.

inherit Inherit all the decorations from the same method in the parent class. This will let the exposed method expose the same template as the overridden method template and keep the same hooks and validation that the parent method had.

The expose decorator registers a number of attributes on the decorated function, but does not actually wrap the function the way TurboGears 1.0 style expose decorators did.

This means that we don’t have to play any kind of special tricks to maintain the signature of the exposed function.

The `exclude_names` parameter is new, and it takes a list of keys that ought to be scrubbed from the dictionary before passing it on to the rendering engine. This is particularly useful for JSON.

The `render_parameters` is also new. It takes a dictionary of arguments that ought to be sent to the rendering engine, like this:

```
render_params={'method': 'xml', 'doctype': None}
```

Expose decorator can be stacked like this:

```
@expose('json', exclude_names='d')
@expose('kid:blogtutorial.templates.test_form',
        content_type='text/html')
@expose('kid:blogtutorial.templates.test_form_xml',
        content_type='text/xml', custom_format='special_xml')
def my_exposed_method(self):
    return dict(a=1, b=2, d="username")
```

The `expose('json')` syntax is a special case. `json` is a rendering engine, but unlike others it does not require a template, and `expose` assumes that it matches `content_type='application/json'`

If you want to declare a desired `content_type` in a url, you can use the mime-type style dotted notation:

```
"/mypage.json" ==> for json
"/mypage.html" ==> for text/html
"/mypage.xml" ==> for xml.
```

If you're doing an http post, you can also declare the desired content type in the accept headers, with standard content type strings.

By default `expose` assumes that the template is for html. All other `content_types` must be explicitly matched to a template and engine.

The last `expose` decorator example uses the `custom_format` parameter which takes an arbitrary value (in this case `'special_xml'`). You can then use the `'use_custom_format'` function within the method to decide which of the `'custom_format'` registered `expose` decorators to use to render the template.

`tg.decorators.override_template` (*view, template*)

Override the template to be used.

Use `override_template` in a controller method in order to change the template that will be used to render the response dictionary dynamically.

The `view` argument is the actual controller method for which you want to replace the template.

The `template` string passed in requires that you include the template engine name, even if you're using the default.

So you have to pass in a template id string like:

```
"genshi:myproject.templates.index2"
```

future versions may make the `genshi:` optional if you want to use the default engine.

```
class tg.decorators.paginate(name, use_prefix=False, items_per_page=10,
                             max_items_per_page=0)
```

Paginate a given collection.

This decorator is mainly exposing the functionality of `webhelpers.paginate()`.

Usage

You use this decorator as follows:

```
class MyController(object):
    @expose()
    @paginate("collection")
    def sample(self, *args):
        collection = get_a_collection()
        return dict(collection=collection)
```

To render the actual pager, use:

```
${tmpl_context.paginators.<name>.pager() }
```

It is possible to have several `paginate()`-decorators for one controller action to paginate several collections independently from each other. If this is desired, don't forget to set the `use_prefix`-parameter to `True`.

Parameters

name the collection to be paginated.

items_per_page the number of items to be rendered. Defaults to 10

max_items_per_page the maximum number of items allowed to be set via parameter. Defaults to 0 (does not allow to change that value).

use_prefix if `True`, the parameters the `paginate` decorator renders and reacts to are prefixed with "`<name>_`". This allows for multi-pagination.

class `tg.decorators.require` (*predicate, denial_handler=None, smart_denial=False*)

Decorator that checks if the specified predicate is met, if it isn't it calls the `denial_handler` to prevent access to the decorated method.

The default authorization denial handler of this protector will flash the message of the unmet predicate with `warning` or `error` as the flash status if the HTTP status code is 401 or 403, respectively.

Parameters

- **predicate** – An object with a `check_authorization(enviro)` method which must raise a `tg.predicates.NotAuthorizedError` if not met.
- **denial_handler** – The callable to be run if authorization is denied (overrides `default_denial_handler` if defined).
- **smart_denial** – A list of response types for which to trigger the smart denial, which will act as an API providing a pass-through `tg.controllers.util.abort()`. If `True`, ('application/json', 'text/xml') will be used.

If called, `denial_handler` will be passed a positional argument which represents a message on why authorization was denied.

Use `allow_only` property of `TGController` for controller-wide authorization.

default_denial_handler (*reason*)

Authorization denial handler for protectors.

`tg.decorators.use_custom_format` (*controller, custom_format*)

Use `use_custom_format` in a controller in order to change the active `@expose` decorator when available.

class `tg.decorators.validate` (*validators=None, error_handler=None, form=None*)

Registers which validators ought to be applied.

If you want to validate the contents of your form, you can use the `@validate()` decorator to register the validators that ought to be called.

Parameters

validators Pass in a dictionary of `FormEncode` validators. The keys should match the form field names.

error_handler Pass in the controller method which should be used to handle any form errors

form Pass in a `ToscaWidget` based form with validators

The first positional parameter can either be a dictionary of validators, a `FormEncode` schema validator, or a callable which acts like a `FormEncode` validator.

class `tg.decorators.with_engine` (*engine_name=None, master_params={}*)
Decorator to force usage of a specific database engine in TurboGears SQLAlchemy BalancedSession.

Parameters

- **engine_name** – ‘master’ or the name of one of the slaves, if is None it will not force any specific engine.
- **master_params** – A dictionary or GET parameters that when present will force usage of the master node. The keys of the dictionary will be the name of the parameters to look for, while the values must be whenever to pop the paramter from the parameters passed to the controller (True/False). If *master_params* is a list then it is converted to a dictionary where the keys are the entries of the list and the value is always True.

class `tg.caching.cached_property` (*func*)
Works like python `@property` but the decorated function only gets executed once, successive accesses to the property will just return the value previously stored into the object.

The `@cached_property` decorator can be executed within a provided context, for example to make the cached property thread safe a Lock can be provided:

```
from threading import Lock
from tg.caching import cached_property

class MyClass(object):
    @cached_property
    def my_property(self):
        return 'Value!'
    my_property.context = Lock()
```

5.2.2 Validation

class `tg.decorators.validate` (*validators=None, error_handler=None, form=None*)
Registers which validators ought to be applied.

If you want to validate the contents of your form, you can use the `@validate()` decorator to register the validators that ought to be called.

Parameters

- **validators** Pass in a dictionary of FormEncode validators. The keys should match the form field names.
- **error_handler** Pass in the controller method which should be used to handle any form errors
- **form** Pass in a ToscaWidget based form with validators

The first positional parameter can either be a dictionary of validators, a FormEncode schema validator, or a callable which acts like a FormEncode validator.

exception `tg.validation.TGValidationError` (*msg, value=None, error_dict=None*)
Invalid data was encountered during validation.

The constructor can be passed a short message with the reason of the failed validation.

5.2.3 Authorization

class `tg.decorators.require` (*predicate, denial_handler=None, smart_denial=False*)
Decorator that checks if the specified predicate it met, if it isn't it calls the `denial_handler` to prevent access to the decorated method.

The default authorization denial handler of this protector will flash the message of the unmet predicate with warning or error as the flash status if the HTTP status code is 401 or 403, respectively.

Parameters

- **predicate** – An object with a `check_authorization(envron)` method which must raise a `tg.predicates.NotAuthorizedError` if not met.
- **denial_handler** – The callable to be run if authorization is denied (overrides `default_denial_handler` if defined).
- **smart_denial** – A list of response types for which to trigger the smart denial, which will act as an API providing a pass-through `tg.controllers.util.abort()`. If True, ('application/json', 'text/xml') will be used.

If called, `denial_handler` will be passed a positional argument which represents a message on why authorization was denied.

Use `allow_only` property of `TGController` for controller-wide authorization.

Built-in predicate checkers.

This is mostly took from `repoze.what.predicates`

This module provides the predicate checkers that were present in the original “identity” framework of TurboGears 1, plus others.

class `tg.predicates.CompoundPredicate` (**predicates*, ***kwargs*)
A predicate composed of other predicates.

class `tg.predicates.All` (**predicates*, ***kwargs*)
Check that all of the specified predicates are met.

Parameters `predicates` – All of the predicates that must be met.

Example:

```
# Grant access if the current month is July and the user belongs to
# the human resources group.
p = All(is_month(7), in_group('hr'))
```

evaluate (*environ*, *credentials*)
Evaluate all the predicates it contains.

Parameters

- **environ** – The WSGI environment.
- **credentials** – The `repoze.what` credentials.

Raises `NotAuthorizedError` If one of the predicates is not met.

class `tg.predicates.Any` (**predicates*, ***kwargs*)
Check that at least one of the specified predicates is met.

Parameters `predicates` – Any of the predicates that must be met.

Example:

```
# Grant access if the currest user is Richard Stallman or Linus
# Torvalds.
p = Any(is_user('rms'), is_user('linus'))
```

evaluate (*environ*, *credentials*)
Evaluate all the predicates it contains.

Parameters

- **environ** – The WSGI environment.
- **credentials** – The `repoze.what` credentials.

Raises `NotAuthorizedError` If none of the predicates is met.

class `tg.predicates.has_all_permissions` (**permissions, **kwargs*)
Check that the current user has been granted all of the specified permissions.

Parameters **permissions** – The names of all the permissions that must be granted to the user.

Example:

```
p = has_all_permissions('view-users', 'edit-users')
```

class `tg.predicates.has_any_permission` (**permissions, **kwargs*)
Check that the user has at least one of the specified permissions.

Parameters **permissions** – The names of any of the permissions that have to be granted to the user.

Example:

```
p = has_any_permission('manage-users', 'edit-users')
```

class `tg.predicates.has_permission` (*permission_name, **kwargs*)
Check that the current user has the specified permission.

Parameters **permission_name** – The name of the permission that must be granted to the user.

Example:

```
p = has_permission('hire')
```

class `tg.predicates.in_all_groups` (**groups, **kwargs*)
Check that the user belongs to all of the specified groups.

Parameters **groups** – The name of all the groups the user must belong to.

Example:

```
p = in_all_groups('developers', 'designers')
```

class `tg.predicates.in_any_group` (**groups, **kwargs*)
Check that the user belongs to at least one of the specified groups.

Parameters **groups** – The name of any of the groups the user may belong to.

Example:

```
p = in_any_group('directors', 'hr')
```

class `tg.predicates.in_group` (*group_name, **kwargs*)
Check that the user belongs to the specified group.

Parameters **group_name** (*str*) – The name of the group to which the user must belong.

Example:

```
p = in_group('customers')
```

class `tg.predicates.is_user` (*user_name, **kwargs*)
Check that the authenticated user's username is the specified one.

Parameters **user_name** (*str*) – The required user name.

Example:

```
p = is_user('linus')
```

class `tg.predicates.is_anonymous` (*msg=None*)
Check that the current user is anonymous.

Example:

```
# The user must be anonymous!
p = is_anonymous()
```

New in version 1.0.7.

class `tg.predicates.not_anonymous` (*msg=None*)
Check that the current user has been authenticated.

Example:

```
# The user must have been authenticated!
p = not_anonymous()
```

5.2.4 Pagination

class `tg.decorators.paginate` (*name*, *use_prefix=False*, *items_per_page=10*,
max_items_per_page=0)

Paginate a given collection.

This decorator is mainly exposing the functionality of `webhelpers.paginate()`.

Usage

You use this decorator as follows:

```
class MyController(object):

    @expose()
    @paginate("collection")
    def sample(self, *args):
        collection = get_a_collection()
        return dict(collection=collection)
```

To render the actual pager, use:

```
${tmpl_context.paginators.<name>.pager() }
```

It is possible to have several `paginate()`-decorators for one controller action to paginate several collections independently from each other. If this is desired, don't forget to set the `use_prefix`-parameter to `True`.

Parameters

name the collection to be paginated.

items_per_page the number of items to be rendered. Defaults to 10

max_items_per_page the maximum number of items allowed to be set via parameter. Defaults to 0 (does not allow to change that value).

use_prefix if `True`, the parameters the `paginate` decorator renders and reacts to are prefixed with "`<name>_`". This allows for multi-pagination.

class `tg.support.paginate.Page` (*collection*, *page=1*, *items_per_page=20*)

TurboGears Pagination support for `@paginate` decorator. It is based on a striped down version of the WebHelpers pagination class This represents a page inside a collection of items

pager (*format='~2~'*, *page_param='page'*, *partial_param='partial'*, *show_if_single_page=False*, *separator=' '*, *onclick=None*, *symbol_first='<<'*, *symbol_last='>>'*, *symbol_previous='<'*, *symbol_next='>'*, *link_attr={'class': 'pager_link'}*, *curpage_attr={'class': 'pager_curpage'}*, *dotdot_attr={'class': 'pager_dotdot'}*, *page_link_template='<a%>%'*, *page_plain_template='<span%>%'*, ***kwargs*)
Return string with links to other pages (e.g. "1 2 [3] 4 5 6 7").

format: Format string that defines how the pager is rendered. The string can contain the following \$-tokens that are substituted by the string.Template module:

- `$first_page`: number of first reachable page
- `$last_page`: number of last reachable page
- `$page`: number of currently selected page
- `$page_count`: number of reachable pages
- `$items_per_page`: maximal number of items per page
- `$first_item`: index of first item on the current page
- `$last_item`: index of last item on the current page
- `$item_count`: total number of items
- `$link_first`: link to first page (unless this is first page)
- `$link_last`: link to last page (unless this is last page)
- `$link_previous`: link to previous page (unless this is first page)
- `$link_next`: link to next page (unless this is last page)

To render a range of pages the token `'~3~'` can be used. The number sets the radius of pages around the current page. Example for a range with radius 3:

```
'1 .. 5 6 7 [8] 9 10 11 .. 500'
```

Default: `'~2~'`

symbol_first String to be displayed as the text for the `%(link_first)s` link above.

Default: `'<<'`

symbol_last String to be displayed as the text for the `%(link_last)s` link above.

Default: `'>>'`

symbol_previous String to be displayed as the text for the `%(link_previous)s` link above.

Default: `'<'`

symbol_next String to be displayed as the text for the `%(link_next)s` link above.

Default: `'>'`

separator: String that is used to separate page links/numbers in the above range of pages.

Default: `' '`

page_param: The name of the parameter that will carry the number of the page the user just clicked on.

partial_param: When using AJAX/AJAH to do partial updates of the page area the application has to know whether a partial update (only the area to be replaced) or a full update (reloading the whole page) is required. So this parameter is the name of the URL parameter that gets set to 1 if the ‘onclick’ parameter is used. So if the user requests a new page through a Javascript action (onclick) then this parameter gets set and the application is supposed to return a partial content. And without Javascript this parameter is not set. The application thus has to check for the existence of this parameter to determine whether only a partial or a full page needs to be returned. See also the examples in this modules docstring.

Default: ‘partial’

Note: If you set this argument and are using a URL generator callback, the callback must accept this name as an argument instead of ‘partial’.

show_if_single_page: if True the navigator will be shown even if there is only one page

Default: False

link_attr (optional) A dictionary of attributes that get added to A-HREF links pointing to other pages. Can be used to define a CSS style or class to customize the look of links.

Example: { ‘style’: ‘border: 1px solid green’ }

Default: { ‘class’: ‘pager_link’ }

curpage_attr (optional) A dictionary of attributes that get added to the current page number in the pager (which is obviously not a link). If this dictionary is not empty then the elements will be wrapped in a SPAN tag with the given attributes.

Example: { ‘style’: ‘border: 3px solid blue’ }

Default: { ‘class’: ‘pager_curpage’ }

dotdot_attr (optional) A dictionary of attributes that get added to the ‘..’ string in the pager (which is obviously not a link). If this dictionary is not empty then the elements will be wrapped in a SPAN tag with the given attributes.

Example: { ‘style’: ‘color: #808080’ }

Default: { ‘class’: ‘pager_dotdot’ }

page_link_template (optional) A string with the template used to render page links

Default: ‘<a%*s*>%*s*’

page_plain_template (optional) A string with the template used to render current page, and dots in pagination.

Default: ‘<span%*s*>%*s*’

onclick (optional) This parameter is a string containing optional Javascript code that will be used as the ‘onclick’ action of each pager link. It can be used to enhance your pager with AJAX actions loading another page into a DOM object.

In this string the variable ‘\$partial_url’ will be replaced by the URL linking to the desired page with an added ‘partial=1’ parameter (or whatever you set ‘partial_param’ to). In addition the ‘\$page’ variable gets replaced by the respective page number.

Note that the URL to the destination page contains a ‘partial_param’ parameter so that you can distinguish between AJAX requests (just refreshing the paginated area of your page) and full requests (loading the whole new page).

[Backward compatibility: you can use ‘%*s*’ instead of ‘\$partial_url’]

jQuery example: “\$(‘#my-page-area’).load(‘\$partial_url’); return false;”

Yahoo UI example:

```
“YAHOO.util.Connect.asyncRequest(‘GET’,$partial_url',{
    success:function(o){YAHOO.util.Dom.get(‘#my-page-area’).innerHTML=o.responseText;}
},null); return false;”
```

scriptaculous example:

```
“new Ajax.Updater(‘#my-page-area’, ‘$partial_url’, {asynchronous:true, evalScripts:true});
return false;”
```

ExtJS example: “Ext.get(‘#my-page-area’).load({url:‘\$partial_url’}); return false;”

Custom example: “my_load_page(\$page)”

Additional keyword arguments are used as arguments in the links.

5.2.5 Configuration

class `tg.configuration.AppConfig` (*minimal=False, root_controller=None*)

Class to store application configuration.

This class should have configuration/setup information that is *necessary* for proper application function. Deployment specific configuration information should go in the config files (e.g. development.ini or deployment.ini).

AppConfig instances have a number of methods that are meant to be overridden by users who wish to have finer grained control over the setup of the WSGI environment in which their application is run.

This is the place to configure custom routes, transaction handling, error handling, etc.

add_auth_middleware (*app, skip_authentication*)

Configure authentication and authorization.

Parameters

- **app** – The TG2 application.
- **skip_authentication** (*bool*) – Should authentication be skipped if explicitly requested? (used by repoze.who-testutil)

add_core_middleware (*app*)

Add support for routes dispatch, sessions, and caching middlewares

Those are all deprecated middlewares and will be removed in future TurboGears versions as they have been replaced by other tools.

add_error_middleware (*global_conf, app*)

Add middleware which handles errors and exceptions.

add_ming_middleware (*app*)

Set up the ming middleware for the unit of work

add_sqlalchemy_middleware (*app*)

Set up middleware that cleans up the sqlalchemy session.

The default behavior of TG 2 is to clean up the session on every request. Only override this method if you know what you are doing!

add_tm_middleware (*app*)

Set up the transaction management middleware.

To abort a transaction inside a TG2 app:

```
import transaction
transaction.doom()
```

By default http error responses also roll back transactions, but this behavior can be overridden by overriding `base_config['tm.commit_veto']`.

add_tosca2_middleware (*app*)

Configure the ToscaWidgets2 middleware.

If you would like to override the way the TW2 middleware works, you might do change your `app_cfg.py` to add something like:

```
from tg.configuration import AppConfig
from tw2.core.middleware import TwMiddleware

class MyAppConfig(AppConfig):

    def add_tosca2_middleware(self, app):

        app = TwMiddleware(app,
            default_engine=self.default_renderer,
            translator=ugettext,
            auto_reload_templates = False
        )

        return app
base_config = MyAppConfig()
```

The above example would always set the template auto reloading off. (This is normally an option that is set within your application's ini file.)

add_tosca_middleware (*app*)

Configure the ToscaWidgets middleware.

If you would like to override the way the TW middleware works, you might do something like:

```
from tg.configuration import AppConfig
from tw.api import make_middleware as tw_middleware

class MyAppConfig(AppConfig):

    def add_tosca2_middleware(self, app):

        app = tw_middleware(app, {
            'toscawidgets.framework.default_view': self.default_renderer,
            'toscawidgets.framework.translator': ugettext,
            'toscawidgets.middleware.inject_resources': False,
        })

        return app

base_config = MyAppConfig()
```

The above example would disable resource injection.

There is more information about the settings you can change in the ToscaWidgets *middleware*. <<http://toscawidgets.org/documentation/ToscaWidgets/modules/middleware.html>>

after_init_config (*conf*)

Override this method to set up configuration variables at the application level. This method will be called

after your configuration object has been initialized on startup. Here is how you would use it to override the default setting of `tg.strict_tmpl_context`

```
from tg.configuration import AppConfig
from tg import config

class MyAppConfig(AppConfig):
    def after_init_config(self):
        config['tg.strict_tmpl_context'] = False

base_config = MyAppConfig()
```

make_load_environment()

Return a `load_environment` function.

The returned `load_environment` function can be called to configure the TurboGears runtime environment for this particular application. You can do this dynamically with multiple nested TG applications if necessary.

register_controller_wrapper(wrapper, controller=None)

Registers a TurboGears controller wrapper.

Controller Wrappers are much like a **decorator** applied to every controller. They receive `tg.configuration.AppConfig` instance as an argument and the next handler in chain and are expected to return a new handler that performs whatever it requires and then calls the next handler.

A simple example for a controller wrapper is a simple logging wrapper:

```
def controller_wrapper(app_config, caller):
    def call(*args, **kw):
        try:
            print 'Before handler!'
            return caller(*args, **kw)
        finally:
            print 'After Handler!'
    return call
```

```
base_config.register_controller_wrapper(controller_wrapper)
```

It is also possible to register wrappers for a specific controller:

```
base_config.register_controller_wrapper(controller_wrapper, controller=RootController.index)
```

register_rendering_engine(factory)

Registers a rendering engine factory.

Rendering engine factories are `tg.renderers.base.RendererFactory` subclasses in charge of creating a rendering engine.

register_wrapper(wrapper, after=None)

Registers a TurboGears application wrapper.

Application wrappers are like WSGI middlewares but are executed in the context of TurboGears and work with abstractions like `Request` and `Response` objects.

See `tg.appwrappers.base.ApplicationWrapper` for complete definition of application wrappers.

The `after` parameter defines their position into the wrappers chain. The default value `None` means they are executed in a middle point, so they run after the TurboGears wrappers like `ErrorPageApplicationWrapper` which can intercept their response and return an error page.

Builtin TurboGears wrappers are usually registered with `after=True` which means they run furthest away from the application itself and can intercept the response of any other wrapper.

Providing `after=False` means the wrapper will be registered near to the application itself (so wrappers registered at default position and with `after=True` will be able to see its response).

`after` parameter can also accept an *application wrapper class*. In such case the registered wrapper will be registered right after the specified wrapper and so will be a little further from the application than the specified one (can see the response of the specified one).

setup_auth()

Override this method to define how you would like the authentication options to be setup for your application.

setup_helpers_and_globals()

Add helpers and globals objects to the config.

Override this method to customize the way that `app_globals` and `helpers` are setup.

setup_ming()

Setup MongoDB database engine using Ming

setup_persistence()

Override this method to define how your application configures its persistence model. the default is to setup sqlalchemy from the configuration file, but you might choose to set up a persistence system other than sqlalchemy, or add an additional persistence layer. Here is how you would go about setting up a ming (mongo) persistence layer:

```
class MingAppConfig(AppConfig):
    def setup_persistence(self):
        self.ming_ds = DataStore(config['mongo.url'])
        session = Session.by_name('main')
        session.bind = self.ming_ds
```

setup_routes()

Setup the default TG2 routes

Override this and setup your own routes maps if you want to use custom routes.

It is recommended that you keep the existing application routing in tact, and just add new connections to the mapper above the `routes_placeholder` connection. Lets say you want to add a `tg` controller `SampleController`, inside the `controllers/samples.py` file of your application. You would augment the `app_cfg.py` in the following way:

```
from routes import Mapper
from tg.configuration import AppConfig

class MyAppConfig(AppConfig):
    def setup_routes(self):
        map = Mapper(directory=config['paths']['controllers'],
                    always_scan=config['debug'])

        # Add a Samples route
        map.connect('/samples/', controller='samples', action=index)

        # Setup a default route for the root of object dispatch
        map.connect('*url', controller='root', action='routes_placeholder')

        config['routes.map'] = map
```

```
base_config = MyAppConfig()
```

setup_sqlalchemy()

Setup SQLAlchemy database engine.

The most common reason for modifying this method is to add multiple database support. To do this you might modify your `app_cfg.py` file in the following manner:

```
from tg.configuration import AppConfig, config
from myapp.model import init_model

# add this before base_config =
class MultiDBAppConfig(AppConfig):
    def setup_sqlalchemy(self):
        '''Setup SQLAlchemy database engine(s)'''
        from sqlalchemy import engine_from_config
        engine1 = engine_from_config(config, 'sqlalchemy.first.')
        engine2 = engine_from_config(config, 'sqlalchemy.second.')
        # engine1 should be assigned to sa_engine as well as your first engine's name
        config['tg.app_globals'].sa_engine = engine1
        config['tg.app_globals'].sa_engine_first = engine1
        config['tg.app_globals'].sa_engine_second = engine2
        # Pass the engines to init_model, to be able to introspect tables
        init_model(engine1, engine2)

#base_config = AppConfig()
base_config = MultiDBAppConfig()
```

This will pull the config settings from your `.ini` files to create the necessary engines for use within your application. Make sure you have a look at *Using Multiple Databases In TurboGears* for more information.

setup_tg_wsgi_app (*load_environment=None*)

Create a base TG app, with all the standard middleware.

load_environment A required callable, which sets up the basic environment needed for the application.

setup_vars A dictionary with all special values necessary for setting up the base wsgi app.

```
class tg.wsgiapp.TGApp (config=None, **kwargs)
```

dispatch (*controller, environ, context*)

Dispatches to a controller, the controller itself is expected to implement the routing system.

Override this to change how requests are dispatched to controllers.

find_controller (*controller*)

Locates a controller by attempting to import it then grab the `SomeController` instance from the imported module.

Override this to change how the controller object is found once the URL has been resolved.

resolve (*environ, context*)

Uses dispatching information found in `environ['wsgiorg.routing_args']` to retrieve a controller name and return the controller instance from the appropriate controller module.

Override this to change how the controller name is found and returned.

setup_app_env (*environ*)

Setup Request, Response and TurboGears context objects.

Is also in charge of pushing TurboGears context into the paste registry and detect test mode. Returns whenever the testmode is enabled or not and the TurboGears context.

setup_pylons_compatibility (*environ, controller*)
Updates environ to be backward compatible with Pylons

5.2.6 WebFlash

Flash messaging system for sending info to the user in a non-obtrusive way

class `tg.flash.TGFlash` (***options*)
Support for flash messages stored in a plain cookie.

Supports both fetching flash messages on server side and on client side through Javascript.

When used from Python itself, the flash object provides a `TGFlash.render()` method that can be used from templates to render the flash message.

When used on Javascript, calling the `TGFlash.render()` provides a `webflash` javascript object which exposes `.payload()` and `.render()` methods that can be used to get current message and render it from javascript.

For a complete list of options supported by Flash objects see `TGFlash.configure()`.

configure (*cookie_name='webflash', default_status='ok', template=<string.Template object at 0x7f0d1dc56510>, js_call='webflash.render()', js_template=<string.Template object at 0x7f0d1dc56550>*)

Flash messages can be configured through `AppConfig` (`app_cfg.base_config`) using the following options:

- `flash.cookie_name` -> Name of the cookie used to store flash messages
- `flash.default_status` -> Default message status if not specified (ok by default)
- `flash.template` -> `string.Template` instance used as the flash template when rendered from server side, will receive `$container_id`, `$message` and `$status` variables.
- `flash.js_call` -> javascript code which will be run when displaying the flash from javascript. Default is `webflash.render()`, you can use `webflash.payload()` to retrieve the message and show it with your favourite library.
- `flash.js_template` -> `string.Template` instance used to replace full javascript support for flash messages. When rendering flash message for javascript usage the following code will be used instead of providing the standard `webflash` object. If you replace `js_template` you must also ensure cookie parsing and delete it for already displayed messages. The template will receive: `$container_id`, `$cookie_name`, `$js_call` variables.

message

Get only current flash message, getting the flash message will delete the cookie.

pop_payload()

Fetch current flash message, status and related information.

Fetching flash message deletes the associated cookie.

render (*container_id, use_js=True*)

Render the flash message inside template or provide Javascript support for them.

`container_id` is the DIV where the messages will be displayed, while `use_js` switches between rendering the flash as HTML or for Javascript usage.

status

Get only current flash status, getting the flash status will delete the cookie.

5.2.7 Rendering

`tg.render_template` (*template_vars*, *template_engine=None*, *template_name=None*, ***kwargs*)

Renders a specific template in current TurboGears context.

Permits to manually render any template like TurboGears would for expositions. It also guarantees that the `before_render_call` and `after_render_call` hooks are called in the process.

Parameters

- **template_vars** (*dict*) – This is the dictionary of variables that should become available to the template. Template vars can also include the `tg_cache` dictionary which enables template caching.
- **template_engine** (*str*) – This is the template engine name, same as specified inside `AppConfig.renderers`.
- **template_name** (*str*) – This is the template to render, can be specified both as a path or using dotted notation if available.

TurboGears injects some additional variables in the template context, those include:

- `tg.config` -> like `tg.config` in controllers
- `tg.flash_obj` -> the flash object, call `render` on it to display it.
- `tg.quote_plus` -> function to perform percentage escaping (`%xx`)
- `tg.url` -> like `tg.url` in controllers
- `tg.identity` -> like `tg.request.identity` in controllers
- `tg.session` -> like `tg.session` in controllers
- `tg.locale` -> Languages of the current request
- `tg.errors` -> Validation errors
- `tg.inputs` -> Values submitted for validation
- `tg.request` -> like `tg.request` in controllers
- `tg.auth_stack_enabled` -> if authentication is enabled or not
- `tg.predicates` -> like `tg.predicates` in controllers
- `tmpl_context` -> like `tg.tmpl_context` in controllers
- `response` -> like `tg.response` in controllers
- `request` -> like `tg.request` in controllers
- `config` -> like `tg.config` in controllers
- `app_globals` -> like `tg.app_globals` in controllers
- `session` -> like `tg.session` in controllers
- `url` -> like `tg.url` in controllers
- `h` -> Your application helpers
- `translator` -> The current gettext translator

- `_` -> like `tg.i18n.ugettext`

Additional variables can be added to every template by a `variable_provider` function inside the application configuration. This function is expected to return a `dict` with any variable that should be added the default template variables. It can even replace existing variables.

```
tg.render.cached_template(template_name, render_func, ns_options=(), cache_key=None,
                           cache_type=None, cache_expire=None, **kwargs)
```

Cache and render a template, took from Pylons

Cache a template to the namespace `template_name`, along with a specific key if provided.

Basic Options

template_name Name of the template, which is used as the template namespace.

render_func Function used to generate the template should it no longer be valid or doesn't exist in the cache.

ns_options Tuple of strings, that should correspond to keys likely to be in the `kwargs` that should be used to construct the namespace used for the cache. For example, if the template language supports the 'fragment' option, the namespace should include it so that the cached copy for a template is not the same as the fragment version of it.

Caching options (uses Beaker caching middleware)

cache_key Key to cache this copy of the template under.

cache_type Valid options are `dbm`, `file`, `memory`, `database`, or `memcached`.

cache_expire Time in seconds to cache this template with this `cache_key` for. Or use 'never' to designate that the cache should never expire.

The minimum key required to trigger caching is `cache_expire='never'` which will cache the template forever seconds with no key.

class `tg.renderers.base.RendererFactory`

Factory that creates one or multiple rendering engines for TurboGears. Subclasses have to be registered with `tg.configuration.AppConfig.register_rendering_engine()` and must implement the `create` method accordingly.

classmethod `create` (*config, app_globals*)

Given the TurboGears configuration and application globals it must create a rendering engine for each one specified into the `engines` list.

It must return a dictionary in the form:

```
{'engine_name': rendering_engine_callable,
 'other_engine': other_rendering_callable}
```

Rendering engine callables are callables in the form:

```
func(template_name, template_vars,
      cache_key=None, cache_type=None, cache_expire=None,
      **render_params)
```

`render_params` parameter will contain all the values provide through `@expose(render_params={})`.

options = {}

Here specify the list of engines for which this factory will create a rendering engine and their options. They must be specified like:

```
engines = {'json': {'content_type': 'application/json'}}
```

Currently only supported option is `content_type`.

with_tg_vars = True

Here specify if turbogears variables have to be injected in the template context before using any of the declared engines. Usually `True` unless engines are protocols (ie JSON).

class `tg.jsonify.JSONEncoder` (***kwargs*)

TurboGears custom JSONEncoder.

Provides support for encoding objects commonly used in TurboGears apps, like:

- SQLAlchemy queries
- Ming queries
- Dates
- Decimals
- Generators

Support for additional types is provided through the `__json__` method that will be called on the object by the JSONEncoder when provided and through the ability to register custom encoder for specific types using `JSONEncoder.register_custom_encoder()`.

configure (*isodates=False, custom_encoders=None, **kwargs*)

JSON encoder can be configured through `AppConfig` (`app_cfg.base_config`) using the following options:

- `json.isodates` -> encode dates using ISO8601 format
- `json.custom_encoders` -> List of tuples (`type, encode_func`) to register custom encoders for specific types.

register_custom_encoder (*objtype, encoder*)

Register a custom encoder for the given type.

Instead of using standard behavior for encoding the given type to JSON, the `encoder` will be used instead. `encoder` must be a callable that takes the object as argument and returns an object that can be encoded in JSON (usually a dict).

`tg.jsonify.encode` (*obj, encoder=None, iterencode=False*)

Return a JSON string representation of a Python object.

`tg.jsonify.encode_iter` (*obj, encoder=None*)

Encode object, yielding each string representation as available.

5.2.8 Request & Response

class `tg.request_local.Request` (*environ, charset=None, unicode_errors=None, decode_param_names=None, **kw*)

WebOb Request subclass

The WebOb `webob.Request` has no charset, or other defaults. This subclass adds defaults, along with several methods for backwards compatibility with `paste.wsgiwrappers.WSGIRequest`.

signed_cookie (*name, secret*)

Extract a signed cookie of `name` from the request

The cookie is expected to have been created with `Response.signed_cookie`, and the `secret` should be the same as the one used to sign it.

Any failure in the signature of the data will result in None being returned.

```
class tg.request_local.Response (body=None, status=None, headerlist=None, app_iter=None, content_type=None, conditional_response=None, **kw)
```

WebOb Response subclass

content

The body of the response, as a `str`. This will read in the entire `app_iter` if necessary.

signed_cookie (*name, data, secret, **kwargs*)

Save a signed cookie with `secret` signature

Saves a signed cookie of the pickled data. All other keyword arguments that `WebOb.set_cookie` accepts are usable and passed to the `WebOb.set_cookie` method after creating the signed cookie value.

5.2.9 Hooks

```
class tg.configuration.hooks.HooksNamespace
```

Manages hooks registrations and notifications

disconnect (*hook_name, func, controller=None*)

Disconnect an hook.

The registered function is removed from the hook notification list.

notify (*hook_name, args=None, kwargs=None, controller=None, context_config=None, trap_exceptions=False*)

Notifies a TurboGears hook.

Each function registered for the given hook will be executed, `args` and `kwargs` will be passed to the registered functions as arguments.

It permits to notify both application hooks:

```
tg.hooks.notify('custom_global_hook')
```

Or controller hooks:

```
tg.hooks.notify('before_render', args=(remainder, params, output),
               controller=RootController.index)
```

notify_with_value (*hook_name, value, controller=None, context_config=None*)

Notifies a TurboGears hook which is expected to return a value.

hooks with values are expected to accept an input value and return a replacement for it. Each registered function will receive as input the value returned by the previous function in chain.

The resulting value will be returned by the `notify_with_value` call itself:

```
app = tg.hooks.notify_with_value('before_config', app)
```

register (*hook_name, func, controller=None*)

Registers a TurboGears hook.

Given an hook name and a function it registers the provided function for that role. For a complete list of hooks provided by default have a look at [Hooks and Wrappers](#).

It permits to register hooks both application wide or for specific controllers:

```
tg.hooks.register('before_render', hook_func, controller=RootController.index)
tg.hooks.register('startup', startup_function)
```

class `tg.appwrappers.base.ApplicationWrapper` (*next_handler, config*)

Basic interface of the TurboGears Application Wrappers.

Application wrappers are like WSGI middlewares but are executed in the context of TurboGears and work with abstractions like Request and Response objects.

Application Wrappers can be registered using `AppConfig.register_wrapper()` which will inject them into the next *TGApp* created.

While they can be any callable, inheriting from this base class is strongly suggested as enables additional behaviours and third party code might depend on them.

Application Wrappers require a `next_handler` which is the next handler to call in the chain and `config` which is the current application configuration.

__call__ (*controller, environ, context*)

This is the actual wrapper implementation.

Wrappers are called for each request with the `controller` in charge of handling the request, the `environ` of the request and the TurboGears `context` of the request.

They should call the `next_handler` (which will accept the same parameters) and return a `tg.request_local.Response` instance which is the request response. Usually they will return the same response object provided by the next handler unless they want to replace it.

A simple logging wrapper might look like:

```
class LogAppWrapper(ApplicationWrapper):
    def __init__(self, handler, config):
        super(LogAppWrapper, self).__init__(handler, config)

    def __call__(self, controller, environ, context):
        print 'Going to run %s' % context.request.path
        return self.next_handler(controller, environ, context)
```

injected

Whenever the Application Wrapper should be injected.

By default all application wrappers are injected into the wrappers chain, you might want to make so that they are injected or not depending on configuration options.

next_handler

The next handler in the chain

5.2.10 Milestones

class `tg.configuration.milestones._ConfigMilestoneTracker` (*name*)

Tracks actions that need to be performed when a specific configuration point is reached and required options are correctly initialized

reach ()

Marks the milestone as reached.

Runs the registered actions. Calling this method multiple times should lead to nothing.

register (*action, persist_on_reset=False*)

Registers an action to be called on milestone completion.

If milestone is already passed action is immediately called

5.2.11 Internationalization

`tg.i18n.set_lang` (*languages*, ***kwargs*)

Set the current language(s) used for translations in current call and session.

languages should be a string or a list of strings. First lang will be used as main lang, others as fallbacks.

`tg.i18n.get_lang` (*all=True*)

Return the current i18n languages used

returns `None` if no supported language is available (no translations are in place) or a list of languages.

In case *all* parameter is `False` only the languages for which the application is providing a translation are returned. Otherwise all the languages preferred by the user are returned.

`tg.i18n.add_fallback` (*lang*, ***kwargs*)

Add a fallback language from which words not matched in other languages will be translated to.

This fallback will be associated with the currently selected language – that is, resetting the language via `set_lang()` resets the current fallbacks.

This function can be called multiple times to add multiple fallbacks.

`tg.i18n.set_request_lang` (*languages*, *ttl=None*)

Set the current request language(s) used for translations without touching the session language.

languages should be a string or a list of strings. First lang will be used as main lang, others as fallbacks.

`tg.i18n.ugettext` (*value*)

Mark a string for translation. Returns the localized unicode string of value.

Mark a string to be localized as follows:

```
_('This should be in lots of languages')
```

`tg.i18n.lazy_ugettext` (**args*, ***kwargs*)

Lazy-evaluated version of the `ugettext` function

Mark a string for translation. Returns the localized unicode string of value.

Mark a string to be localized as follows:

```
_('This should be in lots of languages')
```

`tg.i18n.ungettext` (*singular*, *plural*, *n*)

Mark a string for translation. Returns the localized unicode string of the pluralized value.

This does a plural-forms lookup of a message id. *singular* is used as the message id for purposes of lookup in the catalog, while *n* is used to determine which plural form to use. The returned message is a Unicode string.

Mark a string to be localized as follows:

```
ungettext('There is %(num)d file here', 'There are %(num)d files here',  
n) % {'num': n}
```

`tg.i18n.lazy_ungettext` (**args*, ***kwargs*)

Lazy-evaluated version of the `ungettext` function

Mark a string for translation. Returns the localized unicode string of the pluralized value.

This does a plural-forms lookup of a message id. *singular* is used as the message id for purposes of lookup in the catalog, while *n* is used to determine which plural form to use. The returned message is a Unicode string.

Mark a string to be localized as follows:

```
ungettext('There is %(num)d file here', 'There are %(num)d files here',
          n) % {'num': n}
```

5.2.12 Controller Utilities

Helper functions for controller operation.

URL definition and browser redirection are defined here.

```
tg.controllers.util.url (base_url='/', params=None, qualified=False)
    Generate an absolute URL that's specific to this application.
```

The URL function takes a string (`base_url`) and, appends the `SCRIPT_NAME` and adds parameters for all of the parameters passed into the `params` dict.

```
tg.controllers.util.lurl (base_url=None, params=None)
    Like tg.url but is lazily evaluated.
```

This is useful when creating global variables as no request is in place.

As without a request it wouldn't be possible to correctly calculate the url using the `SCRIPT_NAME` this demands the url resolution to when it is displayed for the first time.

```
tg.controllers.util.redirect (base_url='/', params={}, redirect_with=<class
                             'tg.exceptions.HTTPFound'>, **kwargs)
    Generate an HTTP redirect.
```

The function raises an exception internally, which is handled by the framework. The URL may be either absolute (e.g. `http://example.com` or `/myfile.html`) or relative. Relative URLs are automatically converted to absolute URLs. Parameters may be specified, which are appended to the URL. This causes an external redirect via the browser; if the request is POST, the browser will issue GET for the second request.

```
tg.controllers.util.etag_cache (key=None)
    Use the HTTP Entity Tag cache for Browser side caching
```

If a "If-None-Match" header is found, and equivalent to `key`, then a 304 HTTP message will be returned with the ETag to tell the browser that it should use its current cache of the page.

Otherwise, the ETag header will be added to the response headers.

```
tg.controllers.util.abort (status_code=None, detail='', headers=None, comment=None,
                          passthrough=False, error_handler=False)
    Aborts the request immediately by returning an HTTP exception
```

In the event that the `status_code` is a 300 series error, the `detail` attribute will be used as the Location header should one not be specified in the `headers` attribute.

passthrough When `True` instead of displaying the custom error document for errors or the authentication page for failed authorizations the response will just pass through as is.

Set to `"json"` to send out the response body in JSON format.

error_handler When `True` instead of immediately abort the request it will create a callable that can be used as `@validate error_handler`.

A common case is `abort(404, error_handler=True)` as `error_handler` for validation that retrieves objects from database:

```
from formencode.validators import Wrapper

@validate({'team': Wrapper(to_python=lambda value:
                          Group.query.find({'group_name': value}).one())},
```

```

        error_handler=abort(404, error_handler=True))
    def view_team(self, team):
        return dict(team=team)

```

`tg.controllers.util.auth_force_logout()`
 Forces user logout if authentication is enabled.

`tg.controllers.util.auth_force_login(user_name)`
 Forces user login if authentication is enabled.

As TurboGears identifies users by `user_name` the passed parameter should be anything your application declares being the `user_name` field in models.

`tg.controllers.util.validation_errors_response(*args, **kwargs)`
 Returns a `Response` object with validation errors.

The response will be created with a *412 Precondition Failed* status code and errors are reported in JSON format as response body.

Typical usage is as `error_handler` for JSON based api:

```

@expose('json')
@validate({'display_name': validators.NotEmpty(),
          'group_name': validators.NotEmpty() },
          error_handler=validation_errors_response)
def post(self, **params):
    group = Group(**params)
    return dict(group=group)

```

5.2.13 General Utilities

Utilities

class `tg.util.Bunch`
 A dictionary that provides attribute-style access.

class `tg.util.DottedFileNameFinder`
 this class implements a cache system above the `get_dotted_filename` function and is designed to be stuffed inside the `app_globals`.

It exposes a method named `get_dotted_filename` with the exact same signature as the function of the same name in this module.

The reason is that it uses this function itself and just adds caching mechanism on top.

get_dotted_filename (*template_name*, *template_extension*='.html')

this helper function is designed to search a template or any other file by python module name.

Given a string containing the file/template name passed to the `@expose` decorator we will return a resource useable as a filename even if the file is in fact inside a zipped egg.

The actual implementation is a revamp of the Genshi buffet support plugin, but could be used with any kind a file inside a python package.

@param `template_name`: the string representation of the template name as it has been given by the user on his `@expose` decorator. Basically this will be a string in the form of: "genshi:myapp.templates.somename"
 @type `template_name`: string

@param `template_extension`: the extension we expect the template to have, this MUST be the full extension as returned by the `os.path.splitext` function. This means it should contain the dot. ie: '.html'

This argument is optional and the default value if nothing is provided will be `'html'` @type `template_extension`: string

classmethod `lookup` (*name*, *extension*='html')

Convenience method that permits to quickly get a file by dotted notation.

Creates a `DottedFileNameFinder` and uses it to lookup the given file using dotted notation. As `DottedFileNameFinder` provides a lookup cache, using this method actually disables the cache as a new finder is created each time, for this reason if you have recurring lookups it's better to actually create a dotted filename finder and reuse it.

class `tg.util.LazyString` (*func*, **args*, ***kwargs*)

Behaves like a string, but no instance is created until the string is actually used.

Takes a function which should be a string factory and a set of arguments to pass to the factory. Whenever the string is accessed or manipulated the factory is called to create the actual string. This is used mostly by lazy internationalization.

`tg.util.lazify` (*func*)

Decorator to return a lazy-evaluated version of the original

Applying decorator to a function it will create a `LazyString` with the decorated function as factory.

`tg.util.no_warn` (*f*, **args*, ***kwargs*)

Decorator that suppresses warnings inside the decorated function

class `tg.configuration.utils.GlobalConfigurable`

Defines a configurable TurboGears object with a global default instance.

`GlobalConfigurable` are objects which the user can create multiple instances to use in its own application or third party module, but for which TurboGears provides a default instance.

Common examples are `tg.flash` and the default JSON encoder for which TurboGears provides default instances of `.TGFlash` and `.JSONEncoder` classes but users can create their own.

While user created versions are configured calling the `GlobalConfigurable.configure()` method, global versions are configured by `AppConfig` which configures them when `config_ready` milestone is reached.

configure (***options*)

Expected to be implemented by each object to proceed with actualy configuration.

Configure method will receive all the options whose name starts with `CONFIG_NAMESPACE` (example `json.isodates` has `json.namespace`).

If `CONFIG_OPTIONS` is specified options values will be converted with `coerce_config()` passing `CONFIG_OPTIONS` as the `converters` dictionary.

classmethod `create_global` ()

Creates a global instance which configuration will be bound to `AppConfig`.

`tg.configuration.utils.coerce_config` (*configuration*, *prefix*, *converters*)

Extracts a set of options with a common prefix and converts them.

To extract all options starting with `trace_errors.` from the `conf` dictionary and conver them:

```
trace_errors_config = coerce_config(conf, 'trace_errors.', {
    'smtp_use_tls': asbool,
    'dump_request_size': asint,
    'dump_request': asbool,
    'dump_local_frames': asbool,
    'dump_local_frames_count': asint
})
```

`tg.configuration.utils.coerce_options` (*options, converters*)

Convert some configuration options to expected types.

To replace given options with the converted values in a dictionary you might do:

```
conf.update(coerce_options(conf, {
    'debug': asbool,
    'serve_static': asbool,
    'auto_reload_templates': asbool
}))
```

`tg.configuration.utils.get_partial_dict` (*prefix, dictionary, container_type=<type 'dict'>*)

Given a dictionary and a prefix, return a Bunch, with just items that start with prefix

The returned dictionary will have 'prefix.' stripped so:

```
get_partial_dict('prefix', {'prefix.xyz':1, 'prefix.zyx':2, 'xy':3})
```

would return:

```
{'xyz':1, 'zyx':2}
```

The TurboGears documentation

<i>Tutorials & Documentation</i>	<i>TurboGears CookBook</i>	<i>TurboGears Reference</i>
--------------------------------------	----------------------------	-----------------------------

Getting Started

TurboGears is a Python web framework based on the *ObjectDispatch* paradigm, it is meant to make possible to write both small and concise applications in *Minimal mode* or complex application in *Full Stack mode*.

7.1 Installing TurboGears

TurboGears is meant to run inside python virtualenv and provides its own private index to avoid messing with your system packages and to provide a reliable set of packages that will correctly work together.

```
$ virtualenv tg2env
$ tg2env/bin/pip install tg.devtools
$ source tg2env/bin/activate
(tg2env)$ #now you are ready to work with TurboGears
```

7.2 Single File Application

TurboGears minimal mode makes possible to quickly create single file applications, this makes easy to create simple examples and web services with a minimal set of dependencies.

```
from wsgiref.simple_server import make_server
from tg import expose, TGController, AppConfig

class RootController(TGController):
    @expose()
    def index(self):
        return "<h1>Hello World</h1>"

config = AppConfig(minimal=True, root_controller=RootController())

print "Serving on port 8080..."
httpd = make_server('', 8080, config.make_wsgi_app())
httpd.serve_forever()
```

Play with it on [Runnable](#) or follow the *Minimal Mode Tutorial* to setup your own *Single File Applications*.

7.3 Full Stack Projects

For more complex projects TurboGears provides the so called full stack setup, to manage full stack projects the *GearBox* command is provided.

To try a full stack TurboGears application feel free to *quickstart* one and start playing around:

```
(tg2env) $ gearbox quickstart -n -x example
(tg2env) $ cd example/
(tg2env) $ python setup.py develop
(tg2env) $ gearbox serve
```

Visiting <http://localhost:8080/index> you will see a ready made sample application with a brief introduction to the framework itself.

Explore the *Getting Started* section to get started with TurboGears!

t

tg.configuration, 98
tg.configuration.utils, 220
tg.controllers.util, 218
tg.decorators, 195
tg.flash, 211
tg.i18n, 217
tg.predicates, 56
tg.render, 213
tg.util, 219
tg.validation, 200

Symbols

`_ConfigMilestoneTracker` (class in `tg.configuration.milestones`), 216

`__call__()` (`tg.appwrappers.base.ApplicationWrapper` method), 216

A

`abort()` (in module `tg.controllers.util`), 218

`add_auth_middleware()` (`tg.configuration.AppConfig` method), 206

`add_core_middleware()` (`tg.configuration.AppConfig` method), 206

`add_error_middleware()` (`tg.configuration.AppConfig` method), 206

`add_fallback()` (in module `tg.i18n`), 217

`add_ming_middleware()` (`tg.configuration.AppConfig` method), 206

`add_sqlalchemy_middleware()` (`tg.configuration.AppConfig` method), 206

`add_tm_middleware()` (`tg.configuration.AppConfig` method), 206

`add_tosca2_middleware()` (`tg.configuration.AppConfig` method), 207

`add_tosca_middleware()` (`tg.configuration.AppConfig` method), 207

`AdminConfig` (class in `tgext.admin.config`), 141

`after_init_config()` (`tg.configuration.AppConfig` method), 207

`after_render` (class in `tg.decorators`), 196

`All` (class in `tg.predicates`), 201

`Any` (class in `tg.predicates`), 201

`AppConfig` (class in `tg.configuration`), 206

`ApplicationWrapper` (class in `tg.appwrappers.base`), 215

`auth_force_login()` (in module `tg.controllers.util`), 219

`auth_force_logout()` (in module `tg.controllers.util`), 219

B

`before_call` (class in `tg.decorators`), 196

`before_render` (class in `tg.decorators`), 196

`before_validate` (class in `tg.decorators`), 196

`Bunch` (class in `tg.util`), 219

C

`CacheApplicationWrapper` (class in `tg.appwrappers.caching`), 194

`cached` (class in `tg.decorators`), 196

`cached_property` (class in `tg.caching`), 200

`cached_template()` (in module `tg.render`), 213

`coerce_config()` (in module `tg.configuration.utils`), 220

`coerce_options()` (in module `tg.configuration.utils`), 220

`CompoundPredicate` (class in `tg.predicates`), 201

`configure()` (`tg.configuration.utils.GlobalConfigurable` method), 220

`configure()` (`tg.flash.TGFlash` method), 211

`configure()` (`tg.jsonify.JSONEncoder` method), 214

`content` (`tg.request_local.Response` attribute), 215

`create()` (`tg.renderers.base.RendererFactory` class method), 213

`create_global()` (`tg.configuration.utils.GlobalConfigurable` class method), 220

`CrudRestController` (class in `tgext.crud`), 137

D

`decode_params` (class in `tg.decorators`), 197

`Decoration` (class in `tg.decorators`), 196

`default_denial_handler()` (`tg.decorators.require` method), 199

`disconnect()` (`tg.configuration.hooks.HooksNamespace` method), 215

`dispatch()` (`tg.wsgiapp.TGApp` method), 210

`DottedFileNameFinder` (class in `tg.util`), 219

E

`encode()` (in module `tg.jsonify`), 214

`encode_iter()` (in module `tg.jsonify`), 214

`ErrorHandlerApplicationWrapper` (class in `tg.appwrappers.errorpage`), 195

`ETag`, 67

`etag_cache()` (in module `tg.controllers.util`), 218

`evaluate()` (`tg.predicates.All` method), 201

evaluate() (tg.predicates.Any method), 201
 expose (class in tg.decorators), 197

F

find_controller() (tg.wsgiapp.TGApp method), 210

G

get_dotted_filename() (tg.util.DottedFileNameFinder method), 219
 get_lang() (in module tg.i18n), 217
 get_partial_dict() (in module tg.configuration.utils), 221
 GlobalConfigurable (class in tg.configuration.utils), 220

H

has_all_permissions (class in tg.predicates), 202
 has_any_permission (class in tg.predicates), 202
 has_permission (class in tg.predicates), 202
 HooksNamespace (class in tg.configuration.hooks), 215

I

I18NApplicationWrapper (class in tg.appwrappers.i18n), 194
 in_all_groups (class in tg.predicates), 202
 in_any_group (class in tg.predicates), 202
 in_group (class in tg.predicates), 202
 injected (tg.appwrappers.base.ApplicationWrapper attribute), 216
 is_anonymous (class in tg.predicates), 203
 is_user (class in tg.predicates), 202

J

JSONEncoder (class in tg.jsonify), 214

L

lazify() (in module tg.util), 220
 lazy_ugettext() (in module tg.i18n), 217
 lazy_ungettext() (in module tg.i18n), 217
 LazyString (class in tg.util), 220
 lookup() (tg.util.DottedFileNameFinder class method), 220
 lookup_template_engine() (tg.decorators.Decoration method), 196
 lurl() (in module tg.controllers.util), 218

M

make_load_environment() (tg.configuration.AppConfig method), 208
 message (tg.flash.TGFlash attribute), 211
 MingApplicationWrapper (class in tg.appwrappers.mingflush), 195

N

next_handler (tg.appwrappers.base.ApplicationWrapper attribute), 216

no_warn() (in module tg.util), 220
 not_anonymous (class in tg.predicates), 203
 notify() (tg.configuration.hooks.HooksNamespace method), 215
 notify_with_value() (tg.configuration.hooks.HooksNamespace method), 215

O

options (tg.renderers.base.RendererFactory attribute), 213
 override_template() (in module tg.decorators), 198

P

Page (class in tg.support.paginate), 203
 pager() (tg.support.paginate.Page method), 204
 paginate (class in tg.decorators), 198
 pop_payload() (tg.flash.TGFlash method), 211

R

reach() (tg.configuration.milestones._ConfigMilestoneTracker method), 216
 redirect() (in module tg.controllers.util), 218
 register() (tg.configuration.hooks.HooksNamespace method), 215
 register() (tg.configuration.milestones._ConfigMilestoneTracker method), 216
 register_controller_wrapper() (tg.configuration.AppConfig method), 208
 register_custom_encoder() (tg.jsonify.JSONEncoder method), 214
 register_custom_template_engine() (tg.decorators.Decoration method), 196
 register_rendering_engine() (tg.configuration.AppConfig method), 208
 register_template_engine() (tg.decorators.Decoration method), 196
 register_wrapper() (tg.configuration.AppConfig method), 208
 render() (tg.flash.TGFlash method), 211
 render_template() (in module tg), 212
 RendererFactory (class in tg.renderers.base), 213
 Request (class in tg.request_local), 214
 require (class in tg.decorators), 199
 resolve() (tg.wsgiapp.TGApp method), 210
 Response (class in tg.request_local), 215

S

SessionApplicationWrapper (class in tg.appwrappers.session), 194
 set_lang() (in module tg.i18n), 217
 set_request_lang() (in module tg.i18n), 217
 setup_app_env() (tg.wsgiapp.TGApp method), 210
 setup_auth() (tg.configuration.AppConfig method), 209
 setup_helpers_and_globals() (tg.configuration.AppConfig method), 209

[setup_ming\(\)](#) (tg.configuration.AppConfig method), 209
[setup_persistence\(\)](#) (tg.configuration.AppConfig method), 209
[setup_pylons_compatibility\(\)](#) (tg.wsgiapp.TGApp method), 211
[setup_routes\(\)](#) (tg.configuration.AppConfig method), 209
[setup_sqlalchemy\(\)](#) (tg.configuration.AppConfig method), 210
[setup_tg_wsgi_app\(\)](#) (tg.configuration.AppConfig method), 210
[signed_cookie\(\)](#) (tg.request_local.Request method), 214
[signed_cookie\(\)](#) (tg.request_local.Response method), 215
[status](#) (tg.flash.TGFlash attribute), 211

T

[tg.configuration](#) (module), 98
[tg.configuration.utils](#) (module), 220
[tg.controllers.util](#) (module), 218
[tg.decorators](#) (module), 195
[tg.flash](#) (module), 211
[tg.i18n](#) (module), 217
[tg.predicates](#) (module), 56, 201
[tg.render](#) (module), 213
[tg.util](#) (module), 219
[tg.validation](#) (module), 200
[TGApp](#) (class in tg.wsgiapp), 210
[TGFlash](#) (class in tg.flash), 211
[TGValidationError](#), 200
[TransactionApplicationWrapper](#) (class in tg.appwrappers.transaction_manager), 195

U

[ugettext\(\)](#) (in module tg.i18n), 217
[ungettext\(\)](#) (in module tg.i18n), 217
[url\(\)](#) (in module tg.controllers.util), 218
[use_custom_format\(\)](#) (in module tg.decorators), 199

V

[validate](#) (class in tg.decorators), 199
[validation_errors_response\(\)](#) (in module tg.controllers.util), 219

W

[with_engine](#) (class in tg.decorators), 199
[with_tg_vars](#) (tg.renderers.base.RendererFactory attribute), 214