Whetting the Appetite: Make 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.

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 tg22env
$ . tg22env/bin/activate
```

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

```
(tg22env)$
```

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:

```
(tg22env)$ pip install -i http://tg.gy/current tg.devtools
```

Note

The *-i http://tg.gy/current* 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.

Quickstart

TurboGears2 provides a suite of tools for working with projects by adding several commands to the Python command line tool `paster`. A few will be touched upon in this tutorial. (Check the `../appendices/commandline` 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:

```
(tg22env)$ paster quickstart
```

You'll be prompted for the name of the project, and the name of the python package. Here's what our choices for this tutorial look like:

```
(tg22env)$ paster quickstart
Enter project name: Wiki 20
Enter package name [wiki20]:
Would you prefer to use an alternative template system? (m=mako, j=jinja, k=kajiki,
n=no [default]):
Do you need authentication and authorization in this project? ([yes]/no):
```

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 and so on.
```

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 >= 2.2.0",
    "Genshi",
    "zope.sqlalchemy >= 0.4",
    "repoze.tm2 >= 1.0a5",
    "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 `Wiki-20` 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 `paster serve` command:

```
(tg22env)$ paster serve development.ini --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.

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>
        <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 Documents</a>
      <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')} Join the
Mail List</a>
        for general TG use/topics
      </div>
    </div>

    <div class="row">
      <div class="span4">
        <h3>Code your data model</h3>
        <p> Design your data <code>model</code>, 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 <code>controller</code> methods, Design
your
          <code>templates</code>, 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>
```

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.wiki 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 `paster setup-app` command where your application configuration file is available (usually the root of the project):

```
(tg22env)$ paster setup-app development.ini
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.

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):
        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)

    #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.

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, `wikipage` is an instance of our mapped `Page` class, which was passed in a dictionary by our controller.):

```
<title>${wikipage.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="wikipage.pagename">Page Name Goes Here</span>
```

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

```
<div py:replace="wikipage.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.

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: ${wikipage.pagename}</title>
    </head>
```

6. Change the div that displays the page:

```
    <div py:replace="wikipage.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="${wikipage.pagename}"/>
    <textarea name="data" py:content="wikipage.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.

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.

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.

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"):
    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.

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 .

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.