

Introduction to Pylons

Although Django is currently the most popular webframework for Python, it is by no means your only choice. Where Django grew out of the needs of newsrooms to implement content management solutions rapidly, Pylons grew out of a need to build web applications in environments that may have existing databases to integrate with, and the applications don't fit neatly into the class of applications that are loosely defined in the "content management" space.

Pylons greatest strength is that it takes a best-of-breed approach to constructing its technology stack. Where everything is "built in" with Django and the entire application stack is specifically designed with a single worldview of how applications should be done, Pylons takes precisely the opposite approach. Pylons, the core codebase that lives in the pylons namespace, is remarkably small. With the 0.9.7 release, it's hovering around 5,500 lines of code. Django, by comparison, weighs in at about 125,000 lines of code.

Pylons manages to do this magic by leveraging existing libraries extensively, and the Pylons community works with many other Python projects to develop standard APIs to promote interoperability.

Ultimately, picking Django or Pylons is about deciding which tradeoffs you're willing to make. Although Django is extremely easy to learn because all the documentation is in one place and all the documentation relating to any particular component is always discussed in the context of building a web application, you lose some flexibility when you need to start doing things that are at the margins of what Django was designed for.

For example, in a project we've worked on recently, we needed to interact with a nontrivial database that was implemented in SQL Server 2000. For Django, implementing the SQL Server back-end was quite difficult. There aren't that many web developers using Django on Windows, never mind SQL Server. While the Django ORM is a part of Django, it is also not the core focus of Django. Supporting arbitrary databases is simply not a goal for Django, and rightly so.

Pylons uses SQLAlchemy, which is probably the most powerful database toolkit available in Python. It *None* focuses on database access. The SQL Server back-end was already built in a robust way for CPython, and implementing the extra code for a Jython backend took two days—and this was without seeing any of the code in SQLAlchemy's internals.

That experience alone sold us on Pylons. We don't have to rely on the webframework people being experts in databases. Similarly, we don't have to rely on the database experts to know anything about web templating.

In short, when you have to deal with the non-standard things, Pylons makes a fabulous choice, and, let's be honest, there's almost always non-standard things you're going to have to do.

A Guide for the Impatient

The best way to install Pylons is inside of a virtualenv. Create a new virtualenv for Jython and run `easy_install`:

Listing 15-1.

```
> easy_install "Pylons==0.9.7"
```

Create your application:

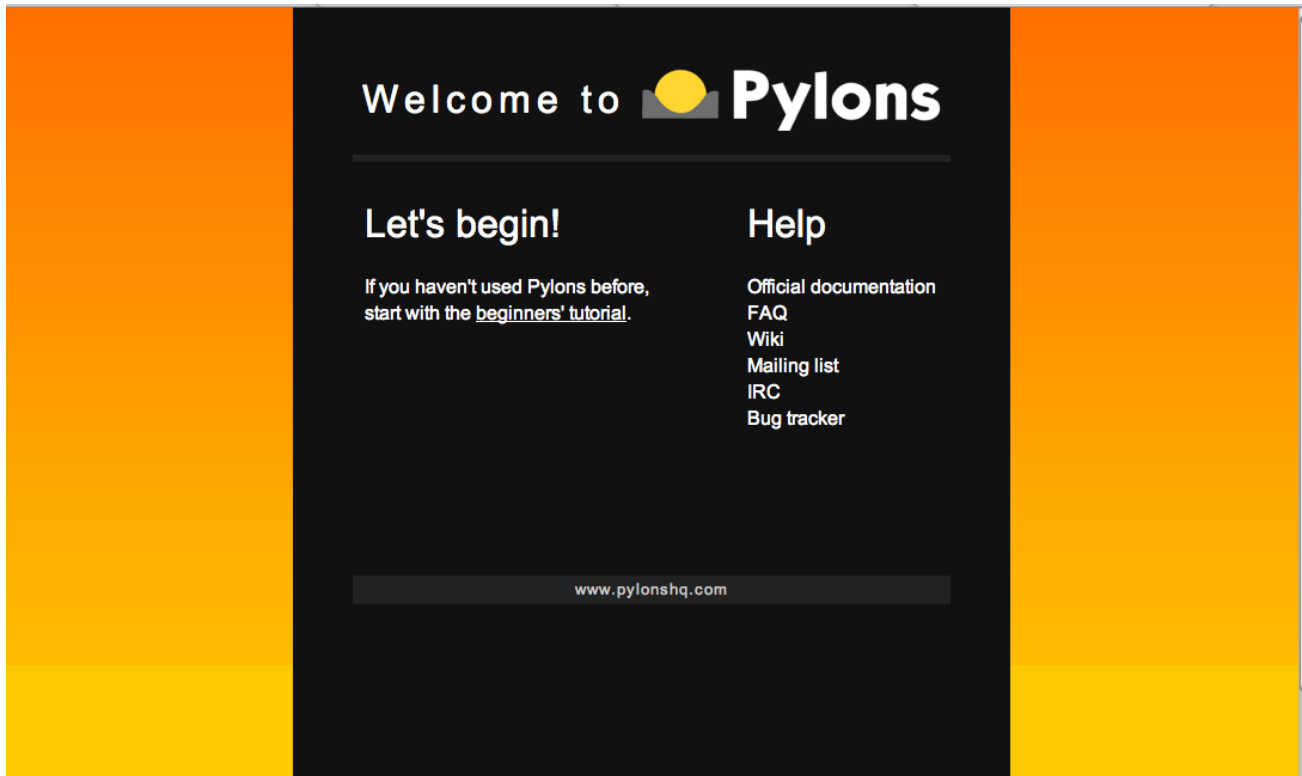
```
> paster create --template=pylons RosterTool
```

After initiating this command, you'll be prompted to enter information about your application. Let's accept the defaults for everything by just hitting Enter at each prompt. After you've entered the default information, your application will be created. You will see a directory named the same as your application created within your current directory. Go into that directory and you will see a series of .py files along with a couple of other files. To configure your development environment, open up the `development.ini` file in a text editor. You will see that there are several parameters in the file that can be changed, including `email_to` for the site administrator's email address, `smtp_server` if you wish to configure mail for application, and many more. For the purposes of this example, we'll leave the default values in the configuration file and continue.

Next, launch the development server using the following command from within the application (*RosterTool*) directory:

```
> paster serve --reload development.ini
```

Open a browser and connect to `http://127.0.0.1:5000/`, and you should see something that looks similar to Figure 15-1.



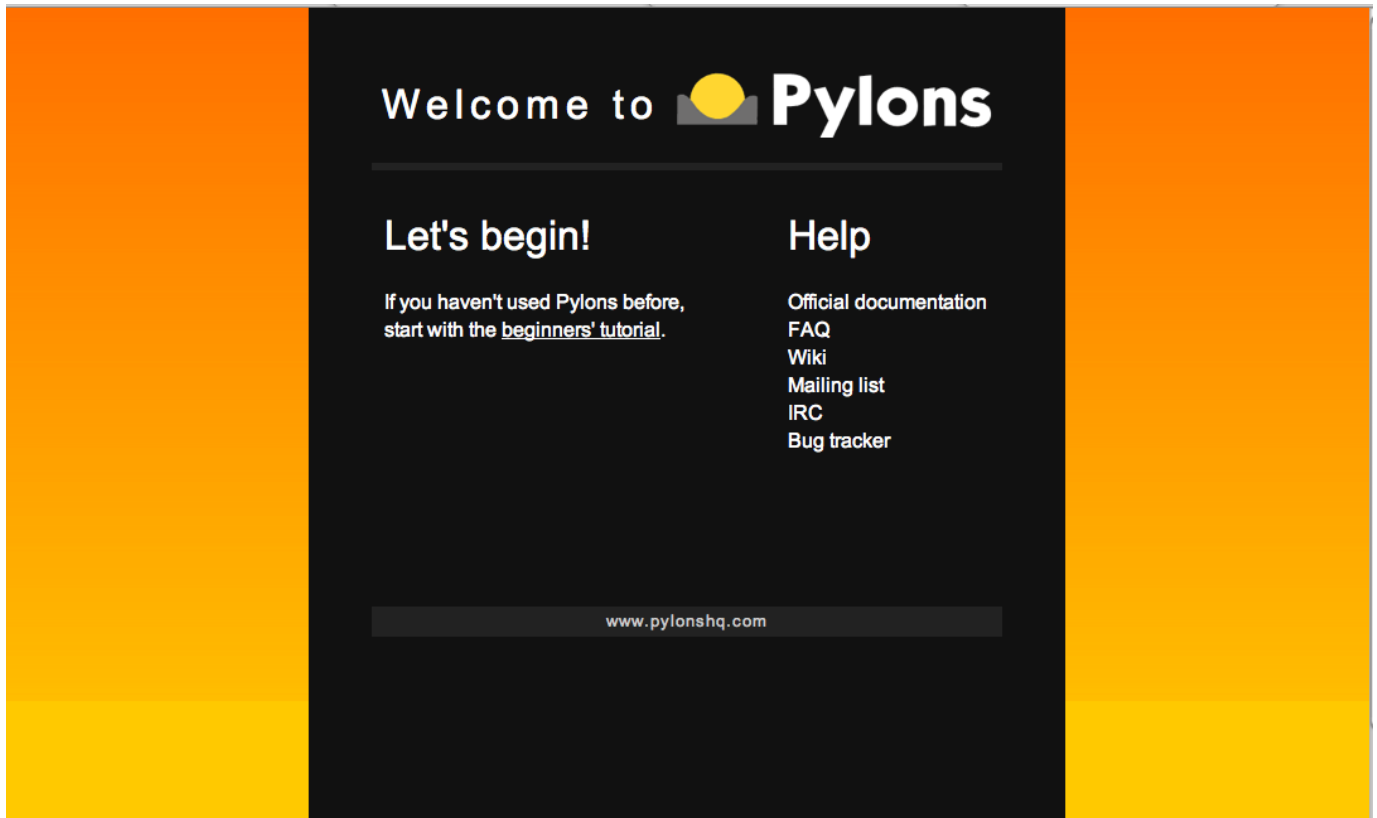
*Figure 15-1. Pylons server *

Now that we have the development server running, it is good to note that we can stop the server at any time by pressing Ctrl+C on the keyboard.

Now we have the base application created, and it is time to add some content. It is easy to create static HTML files for your Pylons application. Let's create a static file named welcome.html and drop it into RosterTool/rosterool/public/ directory.

Listing 15-2.

```
<html>
  <body>Just a static file</body>
</html>
```



You should now be able to load the static content by taking your browser to <http://localhost:5000/welcome.html>.

To create web content that handles requests, we need to create a controller for our application. A controller is a Python module that we map to a URL so that when the URL is visited, the controller is invoked. To add a controller, let's start by initiating the following command:

```
RosterTool/rostertool > paster controller roster
```

Paste will install a directory named "controllers" inside the rostertool directory and install some files in there including a module named roster.py. You can open it up and you'll see a class named "RosterController" and it will have a single method "index." Pylons is smart enough to automatically map a URL to a controller classname and invoke a method. We'll learn more about mapping URLs later in the chapter. To invoke the RosterController's index method, you just need to start the development server again and invoke the following:

```
http://localhost:5000/roster/index
```

Congratulations, you've got your most basic possible web application running now. It handles basic HTTP GET requests and calls a method on a controller and a response comes out. Let's cover each of these pieces in detail now.

A Note about Paste

While you were setting up your toy Pylons application, you probably wondered why Pylons seems to use a command line tool called “paster” instead of something obvious like “pylons.” Paster is actually a part of the Paste set of tools that Pylons uses.

Paste is used to build web applications and frameworks, but most commonly it is used to build web application frameworks like Pylons. Every time you use “paster,” that’s Paste being called. Every time you access the HTTP request and response objects, that’s WebOb, a descendant of Paste’s HTTP wrapper code. Pylons uses Paste extensively for configuration management, testing, basic HTTP handling with WebOb. You would do well to at least skim over the Paste documentation to see what is available in paste, it is available at <http://pythonpaste.org/>.

Pylons MVC

Pylons, like Django and any reasonably sane webframework (or GUI toolkit for that matter) uses the model-view-controller design pattern.

Table 15-1 shows what this maps to in Pylons.

****Table 15-1. **Pylon MVC design pattern mapping**

Component	Implementation
Model	SQLAlchemy (or any other database toolkit you prefer)
View	Mako (or any templating language you prefer)
Controller	Plain Python code

To reiterate, Pylons is about letting you, the application developer, decide on the particular tradeoffs you’re willing to make. If using a template language more similar to the one in Django is better for your web designers, then switch go Jinja2. If you don’t really want to deal with SQLAlchemy, you can use SQLAlchemy, files, a non-relational database, or raw SQL, if you prefer.

Pylons provides tools to help you hook these pieces together in a rational way.

Routes is a library that maps URLs to classes. This is your basic mechanism for dispatching methods whenever your webserver is hit. Routes provides similar functionality to what Django’s URL dispatcher provides.

Webhelpers is the defacto standard library for Pylons. It contains commonly used functions for the web, such as flashing status messages to users, date conversion functions, HTML tag generation, pagination functions, text processing, and the list goes on.

Pylons also provides infrastructure so that you can manipulate things that are particular to web applications including:

- WSGI middleware to add cross-cutting functionality to your application with minimal intrusion into your existing codebase.
- A robust testing framework, including a shockingly good debugger you can use through the web.
- Helpers to enable REST-ful API development so you can expose your application as a programmatic interface.

Later in this chapter, we'll wrap up the hockey roster up in a web application. We'll target a few features:

- Form handling and validation to add new players through the web
- Login and authentication to make sure not anybody can edit our lists
- Add a JSON/REST api so that we can modify data from other tools

In the process, we'll use the interactive debugger from both command line and through the web to directly observe and interact with the state of the running application.

An Interlude into Java's Memory Model

A note about reloading: sometimes if you're doing development with Pylons on Jython, Java will through an OutOfMemory error like this:

Listing 15-3.

```
java.lang.OutOfMemoryError: PermGen space
  at java.lang.ClassLoader.defineClass1(Native Method)
  at java.lang.ClassLoader.defineClass(ClassLoader.java:620)
```

Java keeps track of class definitions in something called the Permanent Generation heap space. This is a problem for Pylons when the HTTP threads are restarted and your classes are reloaded. The old class definitions don't go away; they never get garbage collected. Because Jython is dynamically creating Java classes behind the scenes, each time your development server restarts, you're potentially getting hundreds of new classes loaded into the JVM.

Repeat this several times and it doesn't take long until your JVM has run out of permgen space and it keels over and dies

To modify the permgen heap size, you'll need to instruct Java using some extended command line options. To set the heap to 128M, you'll need to use "-XX:MaxPermSize=128M."

To get this behavior by default for Jython, you'll want to edit your Jython startup script in JYTHON_HOME/bin/jython (or jython.bat) by editing the line that reads:

Listing 15-4.

```
set_JAVA_OPTS=
```

to be

```
set_JAVA_OPTS=-XX:MaxPermSize=128M
```

This shouldn't be a problem in production environments where you're not generating new class definitions during runtime, but it can be quite frustrating during development.

Invoking the Pylons Shell

Yes, we're going to start with testing right away because it will provide you with a way to explore the Pylons application in an interactive way.

Pylons gives you an interactive shell much like Django's. You can start it up with the following commands:

Listing 15-5.

```
RosterTool > jython setup.py egg_info
RosterTool > paster shell test.ini
```

This will yield a nice interactive shell you can start playing with right away. Now let's take a look at those request and response objects in our toy application.

Listing 15-6.

```
RosterTool > paster shell test.ini

Pylons Interactive Shell
Jython 2.5.0 (Release_2_5_0:6476, Jun 16 2009, 13:33:26)
[OpenJDK Server VM (Sun Microsystems Inc.)]

All objects from rostertool.lib.base are available
Additional Objects:
mapper      - Routes mapper object
wsgiapp     - This project's WSGI App instance
app         - paste.fixture wrapped around wsgiapp

>>> resp = app.get('/roster/index')
>>> resp
<Response 200 OK 'Hello World'>
>>> resp.req
<Request at 0x43 GET http://localhost/roster/index>
```

Pylons lets you actually run requests against the application and play with the resulting response. Even for something as "simple" as the HTTP request and response, Pylons uses a

library to provide convenience methods and attributes to make your development life easier. In this case, it's WebOb.

The request and the response objects both have literally dozens of attributes and methods that are provided by the framework. You will almost certainly benefit if you take time to browse through WebOb's documentation, which is available at <http://pythonpaste.org/webob/>.

Here's four attributes you really have to know to make sense of the request object. The best thing to do is to try playing with the request object in the shell.

request.GET

GET is a special dictionary of the variables that were passed in the URL. Pylons automatically converts URL arguments that appear multiple times into discrete key value pairs.

Listing 15-7.

```
>>> resp = app.get('/roster/index?foo=bar&x=42&x=50')
>>> resp.req.GET
UnicodeMultiDict([('foo', u'bar'), ('x', u'42'), ('x', u'50')])
>>> resp.req.GET['x']
u'50'
>>> resp.req.GET.getall('x')
[u'42', u'50']
```

Note how you can get either the last value or the list of values depending on how you choose to fetch values from the dictionary. This can cause subtle bugs if you're not paying attention.

request.POST

POST is similar to GET, but appropriately: it only returns the variables that were sent up during an HTTP POST submission.

request.params

Pylons merges all the GET and POST data into a single MultiValueDict. In almost all cases, this is the one attribute that you really want to use to get the data that the user sent to the server.

request.headers

This dictionary provides all the HTTP headers that the client sent to the server.

Context Variables and Application Globals

Most web frameworks provide a request scoped variable to act as a bag of values. Pylons is no exception: whenever you create a new controller with paste, it will automatically import an attribute 'c' which is the context variable.

This is one aspect of Pylons that we've found to be frustrating. The 'c' attribute is code generated as an import when you instruct paste to build you a new controller. The 'c' value is *None* an attribute of your controller—Pylons has special global threadsafe variables, this is just one of them. You can store variables that you want to exist for the duration of the request in the context. These values won't persist after the request/response cycle has completed, so don't confuse this with the session variable.

The other global variable you'll end up using a lot is pylons.session. This is where you'll store variables that need to persist over the course of several request/response cycles. You can treat this variable as a special dictionary: just use standard Python dictionary syntax and Pylons will handle the rest.

Routes

Routes is much like Django's URL dispatcher. It provides a mechanism for you to map URLs to controllers classes and methods to invoke.

Generally, we find that Routes makes a tradeoff of less URL matching expressiveness in exchange for simpler reasoning about which URLs are directed to a particular controller and method. Routes doesn't support regular expressions, just simple variable substitution.

A typical route will look something like this

```
map.connect('/{mycontroller}/{someaction}/{var1}/{var2}')
```

This route would find the controller called "mycontroller" (note the casing of the class) and invoke the "someaction" method on that object. Variables var1 and var2 would be passed in as arguments.

The connect() method of the map object will also take in optional arguments to fill in default values for URLs that do not have enough URL-encoded data in them to properly invoke a method with the minimum required number of arguments. The front page is an example of this; let's try connecting the frontpage to the Roster.index method.

Edit RosterTool/rosterool/config/routing.py so that there are 3 lines after #CUSTOM_ROUTES_HERE that should read like the following

Listing 15-8.

```
map.connect('/', controller='roster',action='index') ---
map.connect('/{action}/{id}/',controller='roster') ---
```

```
map.connect('/add_player/', controller='roster', action='add_player')
```

While this *None* like it should work, you can try running paster server: it won't.

By default, Pylons always tries to serve static content before searching for controllers and methods to invoke. You'll need to go to `RosterTool/rostertool/public` and delete the "index.html" file that paster installed when you first created your application. If you wanted to change the default implementation, you could tweak the `middleware*.py*` module to your liking.

Load <http://localhost:5000/> again in your browser—the default index.html should be gone and you should now get your response from the controller method that is mapped to index, in this case, you should see "Hello World."

Controllers and Templates

Leveraging off of the Table model we defined in Chapter 12, let's create the hockey roster, but this time using the PostgreSQL database. We'll assume that you have a PostgreSQL installation running that allows you to create new databases. You can also use a different database if you choose by simply creating a different engine with SQLAlchemy. For more details, please visit the [documentation about creating database engines](http://www.sqlalchemy.org/docs/05/dbengine.html) at <http://www.sqlalchemy.org/docs/05/dbengine.html>.

Begin by opening up a Pylons interactive shell and typing the following commands:

Listing 15-9.

```
>>> from sqlalchemy import *
>>> from sqlalchemy.schema import Sequence
>>> db =
create_engine('postgresql+zxjdbc://myuser:mypass@localhost:5432/mydb')
>>> connection = db.connect()
>>> metadata = MetaData()
>>> player = Table('player', metadata,
...     Column('id', Integer, primary_key=True),
...     Column('first', String(50)),
...     Column('last', String(50)),
...     Column('position', String(30)))
>>> metadata.create_all(engine)
```

Now let's wire the data up to the controllers, display some data, and get basic form handling working. We're going to create a basic CRUD (create, read, update, delete) interface to the sqlalchemy model. Because of space constraints, this HTML is going to be very basic; but you'll get a taste of how things fit together.

Paste doesn't just generate a stub for your controller—it will also code generate an empty functional test case in `rostertool/tests/functional/` as `test_roster.py`. We'll visit testing shortly.

Controllers are really where the action occurs in Pylons. This is where your application will take data from the database and prepare it for a template to render it as HTML. Let's put the list of all players on the front page of the site. We'll implement a template to render the list of all players. Then, we'll implement a method in the controller to override the `index()` method of `Roster` to use SQLAlchemy to load the records from disk and send them to the template.

Along the way, we'll touch on template inheritance so that you can see how you can save keystrokes by subclassing your templates in Mako.

First, let's create two templates, `base.html` and `list_players.html` in the `rostertool/templates` directory.

Listing 15-10. base.html

```
<html>
  <body>
    <div class="header">
      ${self.header()}
    </div>

    ${self.body()}
  </body>
</html>

<%def name="header()">
  <h1>${c.page_title}</h1>
  <% messages = h.flash.pop_messages() %>
  % if messages:
  <ul id="flash-messages">
    % for message in messages:
    <li>${message}</li>
    % endfor
  </ul>
  % endif
</%def>
```

Listing 15-11. list_players.html

```
<%inherit file="base.html" />
<table border="1">
  <tr>
    <th>Position</th><th>Last name</th><th>First name</th><th>Edit</th>
  </tr>
  % for player in c.players:
  ${makerow(player)}
  % endfor
</table>

<h2>Add a new player</h2>
${h.form(h.url_for(controller='roster', action='add_player'), method='POST')}
  ${h.text('first', 'First Name')} <br />
  ${h.text('last', 'Last Name')} <br />
```

```

    ${h.text('position', 'Position')} <br />
    ${h.submit('add_player', "Add Player")}
    ${h.end_form()}

<%def name="makerow(row)">
<tr>
    <td>${row.position}</td>\
    <td>${row.last}</td>\
    <td>${row.first}</td>\
    <td><a href="${h.url_for(controller='roster', action='edit_player',
id=row.id)}">Edit</a></td>\
</tr>
</%def>

```

There's quite a bit going on here. The base template lets Mako define a boilerplate set of HTML that all pages can reuse. Each section is defined with a `<%def name="block()"` section, and the blocks are overloaded in the subclassed templates. In effect, Mako lets your page templates look like objects with methods that can render subsections of your pages.

The `list_players.html` template has content that is immediately substituted into the `self.body()` method of the base template. The first part of our body uses our magic context variable 'c'. Here, we're iterating over each of the players in the database and rendering them into a table as a row. Note here that we can use the Mako method syntax to create a method called "makerow" and invoke it directly within our template.

Note

Mako provides a rich set of functions for templating. We're only going to use the most basic parts of Mako: inheritance, variable substitution, and loop iteration to get the toy application working. I strongly suggest you dive into the Mako documentation to discover features and get a better understanding of how to use the template library.

Next, we add in a small form to create new players. The trick here is to see that the form is being generated programmatically by helper functions. Pylons automatically imports `YOURPROJECT/lib/helpers` (in our case, `rostertool.lib.helpers`) as the 'h' variable in your template. The helpers module typically imports functions from parts of Pylons or a dependent library to allow access to those features from anywhere in the application. Although this seems like a violation of "separation of concerns," look at the template and see what it buys us: we get fully decoupled URLs from the particular controller and method that need to be invoked. The template uses a special routes function "url_for" to compute the URL that would have been mapped for a particular controller and method. The last part of our `base.html` file contains code to display alert messages.

Let's take a look at our `rostertool.lib.helpers` module now.

Listing 15-12.

```

from routes import url_for
from webhelpers.html.tags import *
from webhelpers.pyramidlib import Flash as _Flash

# Send alert messages back to the user
flash = _Flash()

```

Here, we're importing the `url_for` function from `routes` to do our URL reversal computations. We import HTML tag generators from the main `html.tags` helper modules and we import `Flash` to provide alert messages for our pages. We'll show you how flash messages are used when we cover the controller code in more detail in the next couple of pages.

Now, create a controller with `paster` (you've already done this if you were impatient at the beginning of the chapter).

Note

If you have already created the controller using `thequickstart` at the beginning of the chapter, you will need to add the SQLAlchemy configuration to the `development*.ini*` file by adding the following line to the file.

```
sqlalchemy.url= postgresql+zxjdbc://dbuser:dbpassword@dbhost:port/dbname
```

Listing 15-13.

```

$ cd ROSTERTOOL/rosterool
$ paster controller roster

```

Next, we need to add the metadata for our database table to the `RosterTool/rosterool/model/__init__.py` module. To do so, change the file so that it reads as follows:

Listing 15-14.

```

"""The application's model objects"""
import sqlalchemy as sa
from sqlalchemy import orm, schema, types

from rosterool.model import meta

def init_model(engine):
    """Call me before using any of the tables or classes in the model"""
    ## Reflected tables must be defined and mapped here
    #global reflected_table
    #reflected_table = sa.Table("Reflected", meta.metadata, autoload=True,
    #                             autoload_with=engine)
    #orm.mapper(Reflected, reflected_table)
    #

```

```

meta.Session.configure(bind=engine)
meta.engine = engine

metadata = schema.MetaData()

# Create the metadata for the player table, and assign it to player_table
player_table = schema.Table('player', metadata,
    schema.Column('id', types.Integer, primary_key=True),
    schema.Column('first', types.Text(), nullable=False),
    schema.Column('last', types.Text(), nullable=False),
    schema.Column('position', types.Text(), nullable=False),
)

# Create a class to be used for mapping the player_table object
class Player(object):
    pass

# Map the Player class to the player table object, we can now refer to the
# player_table using Player
orm.mapper(Player, player_table)

```

Note that we are creating the proper metadata for mapping to the player database table. We then create an empty Player class object and later use the `orm.*mapper*` to map the metadata to the empty Player object. We can now use the Player object to work with our database table.

Next, we should alter the index method that is created inside the RosterController class. We will add an import to bring in the `meta` and `Player` objects, and change the `index` function so that it queries the list of players in the database. In the end, the `index` function should read as follows:

Listing 15-15.

```

from rostertool.model import meta, Player
...
def index(self):
    session = meta.Session()
    c.page_title = 'Player List'
    c.players = session.query(Player).all()
    return render('list_players.html')

```

This code is fairly straightforward; we are simply using a SQLAlchemy session to load all the Player objects from disk and assigning to the special context variable `c`. Pylons is then instructed to render the `list_player.html` file.

The context should be your default place to place values you want to pass to other parts of the application. Note that Pylons will automatically bind in URL values to the context so while you can grab the form values from `self.form_result`, you can also grab URL values from the context.

You should be able to run the debug webserver now and you can get to the front page to load an empty list of players. Startup your debug webserver as you did at the beginning of this

chapter and go to <http://localhost:5000/> to see the page load with your list of players (currently an empty list).

Now we need to get to the meaty part where we can start create, edit, and delete players. We'll make sure that the inputs are at least minimally validated, errors are displayed to the user, and that alert messages are properly populated.

First, we need a page that shows just a single player and provides buttons for edit and delete.

Listing 15-16.

```
<%inherit file="base.html" />

<h2>Edit player</h2>
${h.form(h.url_for(controller='roster', action='save_player',
id=c.player.id), method='POST')}
    ${h.hidden('id', c.player.id)} <br />
    ${h.text('first', c.player.first)} <br />
    ${h.text('last', c.player.last)} <br />
    ${h.text('position', c.player.position)} <br />
    ${h.submit('save_player', "Save Player")}
${h.end_form()}

${h.form(h.url_for(controller='roster', action='delete_player',
id=c.player.id), method='POST')}
    ${h.hidden('id', c.player.id)} <br />
    ${h.hidden('first', c.player.first)} <br />
    ${h.hidden('last', c.player.last)} <br />
    ${h.hidden('position', c.player.position)} <br />
    ${h.submit('delete_player', "Delete Player")}
${h.end_form()}
```

This template assumes that there is a `player` value assigned to the context and, not surprisingly, it's going to be a full-blown instance of the `Player` object that we first saw in Chapter 12. The helper functions let us define our HTML form using web helper tag generation functions. This means you won't have to worry about escaping characters or remembering the particular details of the HTML attributes. The helper tag functions will do sensible things by default. The `h` is a default template variable that refers to the repository of helper functions.

We've set up the edit and delete forms to point to different URLs. You might want to conserve URLs, but having discrete URLs for each action has advantages, especially for debugging. You can trivially view which URLs are being hit on a web server by reading log files. Seeing the same kind of behavior if the URLs are the same, but the behavior is dictated by some form value, well, that's a whole lot harder to debug. It's also a lot harder to setup in your controllers because you need to dispatch the behavior on a per-method level. Why not just have separate methods for separate behavior? Everybody will thank you for it when they need to debug your code in the future.

Before we create our controller methods for create, edit and delete, we'll create a formencode schema to provide basic validation. Again, Pylons doesn't provide validation behavior; it just leverages another library to do so. Add the following class to `rosterool/controllers/roster.py`:

Listing 15-17.

```
class PlayerForm(formencode.Schema):
    # You need the next line to drop the submit button values
    allow_extra_fields=True

    first = formencode.validators.String(not_empty=True)
    last = formencode.validators.String(not_empty=True)
    position = formencode.validators.String(not_empty=True)
```

This simply provides basic string verification on our inputs. Note how this doesn't provide any hint as to what the HTML form looks like or that it's HTML at all. FormEncode can validate arbitrary Python dictionaries and return errors about them.

We're just going to show you the `add` method, and the `edit_player` methods. You should try to implement the `save_player` and `delete_player` methods to make sure you understand what's going on here. First, add the import for the `validate` decorator. Next, add the `add*_player*` and `edit_player` functions to the `RosterController` class.

Listing 15-18.

```
from pylons.decorators import validate

@validate(schema=PlayerForm(), form='index', post_only=False, on_get=True)
def add_player(self):
    first = self.form_result['first']
    last = self.form_result['last']
    position = self.form_result['position']
    session = meta.Session()
    if session.query(Player).filter_by(first=first, last=last).count() > 0:
        h.flash("Player already exists!")
        return h.redirect_to(controller='roster')
    player = Player(first, last, position)
    session.add(player)
    session.commit()
    return h.redirect_to(controller='roster', action='index')

def edit_player(self, id):
    session = meta.Session()
    player = session.query(Player).filter_by(id=id).one()
    c.player = player
    return render('edit_player.html')
```

A couple of notes here. The `edit_player` function is passed the `id` attribute directly by Routes. In the `edit_player` method, `player` is assigned to the context, but the context is never explicitly

passed into the template renderer. Pylons is going to automatically take the attributes bound to the context and write them into the template and render the HTML output. The *c*variable* is automatically available in the template namespace much like the **h* variable as discussed previously.

With the `add_player` method, we're using the `validate` decorator to enforce the inputs against the `PlayerForm`. In the case of error, the `form` attribute of the decorator is used to load an action against the current controller. In this case, `index`, so the front page loads.

The SQLAlchemy code should be familiar to you if you have already gone through Chapter 12. The last line of the `add_player` method is a redirect to prevent problems with hitting reload in the browser. Once all data manipulation has occurred, the server redirects the client to a results page. In the case that a user hits reload on the result page, no data will be mutated.

Here are the signatures of the remaining methods you'll need to implement to make things work:

- `save_player(self):`
- `delete_player(self):`

If you get stuck, you can always consult the working sample code on the book's web site.

Adding a JSON API

JSON integration into Pylons is very straight forward. The steps are roughly the same as adding controller methods for plain HTML views. You invoke `paste`, `paste` then generates your controller stubs and test stubs, you add in some routes to wire controllers to URLs and then you just fill in the controller code.

Listing 15-19.

```
$ cd ROSTERTOOL_HOME/rostartool
$ paster controller api
```

Pylons provides a special `@jsonify` decorator which will automatically convert Python primitive types into JSON objects. It will *not* convert the POST data into an object though; that's your responsibility. Adding a simple read interface into the player list requires only adding a single method to your `ApiController`:

Listing 15-20.

```
@jsonify
def players(self):
    session = Session()
    players = [{'first': p.first,
                'last': p.last,
                'position': p.position,
```

```
        'id': p.id} for p in session.query(Player).all()]
return players
```

Adding a hook so that people can POST data to your server in JSON format to create new player is almost as easy.

Listing 15-21.

```
import simplejson as json

@jsonify def add_player(self):

obj = json.loads(request.body) schema = PlayerForm() try:

form_result = schema.to_python(obj)
except formencode.Invalid, error:
    response.content_type = 'text/plain' return 'Invalid: '+unicode(error)
else:

    session = Session() first, last, position = obj['first'], obj['last'], obj['position']
    if session.query(Player).filter_by(last=last, first=first,

        position=position).count() == 0:
        session.add(Player(first, last, position)) session.commit() return
        {'result': 'OK'}

    else:
        return {'result': 'fail', 'msg': 'Player already exists'}
```

Unit Testing, Functional Testing, and Logging

One of our favorite features in Pylons is its rich set of tools for testing and debugging. It even manages to take social networking, turn it upside-down, and make it into a debugger feature. We'll get to that shortly.

The first step to knowing how to test code in Pylons is to familiarize yourself with the nose testing framework. Nose makes testing simple by getting out of your way. There are no classes to subclass, just start writing functions that start with the word test and nose will run them. Write a class that has test prefixed in the name and nose will treat it as a suite of tests running each method that starts with test. For each test method, nose will execute the setup() method just prior to executing your test and nose will execute the teardown() method after your test case.

Best of all, nose will automatically hunt down anything that looks like a test and will run it for you. There is no complicated chain of test cases you need to organize in a tree. The computer will do that for you.

Let's take a look at your first test case: we'll just instrument the model, in this case SQLAlchemy. Because the model layer has no dependency on Pylons, this effectively tests only your SQLAlchemy model code.

In `ROSTERTOOL_HOME/rosterool/tests`, create a module called `test_models.py` with the following content

Listing 15-22.

```
from rosterool.model import Player, Session, engine

class TestModels(object):

    def setup(self):
        self.cleanup()

    def teardown(self):
        self.cleanup()

    def cleanup(self):
        session = Session()
        for player in session.query(Player):
            session.delete(player)
        session.commit()

    def test_create_player(self):
        session = Session()
        player1 = Player('Josh', 'Juneau', 'forward')
        player2 = Player('Jim', 'Baker', 'forward')
        session.add(player1)
        session.add(player2)

        # But 2 are in the session, but not in the database
        assert 2 == session.query(Player).count()
        assert 0 == engine.execute("select count(id) from
player").fetchone()[0]
        session.commit()

        # Check that 2 records are all in the database
        assert 2 == session.query(Player).count()
        assert 2 == engine.execute("select count(id) from
player").fetchone()[0]
```

Before we can run the tests, we'll need to edit the model module a little so that the models know to look up the connection URL from Pylons configuration file. In your `test.ini`, add a line setting the `sqlalchemy.url` setting to point to your database in the `[app:main]` section.

You should have a line that looks something like this:

Listing 15-23.

```
[app:main]
use = config:development.ini
sqlalchemy.url =
postgresql+zxjdbc://username:password@localhost:5432/mydb
```

Now edit the model file so that the `create_engine` call uses that configuration. This is as simple as importing config from pylons and doing a dictionary lookup. The two lines you want are

Listing 15-24.

```
from pylons import config
engine = create_engine(config['sqlalchemy.url'])
```

and that's it. Your model will now lookup your database connection string from Pylons. Even better, nose will know how to use that configuration as well.

From the command line, you can run the tests from `ROSTERTOOL_HOME` like this now:

Listing 15-25.

```
ROSTERTOOL_HOME $ nosetests rostertool/tests/test_models.py
.
-----
Ran 1 test in 0.502s
```

Perfect! To capture stdout and get verbose output, you can choose to use the `-sv` option. Another nice option is `-pdb-failures`, which will drop you into the debugger on failures. Nose has its own active community of developers. You can get plug-ins to do coverage analysis and performance profiling with some of the plug-ins. Use `nosetests --help` for a list of the options available for a complete list.

Due to the nature of Pylons and its decoupled design, writing small unit tests to test each little piece of code is very easy. Feel free to assemble your tests any which way you want. Just want to have a bunch of test functions? Great! If you need to have setup and teardown and writing a test class makes sense, then do so.

Testing with nose is a joy you aren't forced to fit into any particular structure with respect to where your tests must go so that they will be executed. You can organize your tests in a way that makes the most sense to *you*.

That covers basic unit testing, but suppose we want to test the JSON interface to our hockey roster. We really want to be able to invoke GET and POST on the URLs to make sure that URL routing is working as we expect. We want to make sure that the content-type is properly set to application/x-json. In other words, we want to have a proper functional test, a test that is not as fine grained as a unit test.

The prior exposure to the app object when we ran the paste shell should give you a rough idea of what is required. In Pylons, you can instrument your application code by using a TestController. Lucky for you, Pylons has already created one for you in your `app/tests/` directory. Just import it, subclass it and you can start using the app object just like you did inside of the shell.

Let's take a look at a functional test in detail now. Here's a sample you can save into `rosterool/tests/functional/test_api.py`:

Listing 15-26.

```
from rosterool.tests import *
import simplejson as json
from rosterool.model.models import Session, Player

class TestApiController(TestController):
    # Note that we're using subclasses of unittest.TestCase so we need
    # to be careful with setup/teardown camel casing unlike nose's
    # default behavior

    def setUp(self):
        session = Session()
        for player in session.query(Player):
            session.delete(player)
        session.commit()

    def test_add_player(self):
        data = json.dumps({'first': 'Victor',
                          'last': 'Ng',
                          'position': 'Goalie'})
        # Note that the content-type is set in the headers to make
        # sure that paste.test doesn't URL encode our data
        response = self.app.post(url(controller='api',
action='add_player'),
                                params=data,
                                headers={'content-type': 'application/x-json'})
        obj = json.loads(response.body)
        assert obj['result'] == 'OK'

        # Do it again and fail
        response = self.app.post(url(controller='api',
action='add_player'),
                                params=data,
                                headers={'content-type': 'application/x-json'})
        obj = json.loads(response.body)
        assert obj['result'] <> 'OK'
```

There's a minor detail which you can easily miss when you're using the `TestController` as your superclass. First off, `TestController` is a descendant of `unittest.TestCase` from the standard Python unit test library. Nose will not run `setup` and `teardown` methods on `TestCase` subclasses. Instead, you'll have to use the camel case names that `TestCase` uses.

Reading through the test case should show you how much detail you can be exposed. All your headers are exposed, the response content is exposed; indeed, the HTTP response is completely exposed as an object for you to inspect and verify.

So great, now we can run small unit tests, bigger functional tests; let's take a look at the debugging facilities provided through the web.

Consider what happens with most web application stacks when an error occurs. Maybe you get a stack trace, maybe you don't. If you're lucky, you can see the local variables at each stack frame like Django does. Usually though, you're out of luck if you want to interact with the live application as the error is occurring.

Eventually, you may locate the part of the stack trace that triggered the error, but the only way of sharing that information is through either the mailing lists or by doing a formal patch against source control. Let's take a look at an example of that.

We're going to startup our application in development mode. We're also going to intentionally break some code in the controller to see the stack trace. But first, we'll need to put some data into our app.

Add a `sqlalchemy.url` configuration line to the `development.ini` file as you did in the `test.ini` configuration, and let's startup the application in development mode. We're going to have the server run so that any code changes on the file system are automatically detected and the code is reloaded

```
$ paster serve development.ini --reload
```

We'll add a single player John Doe as a center, and save the record. Next, let's intentionally break some code to trigger the debugger. Modify the `RosterController`'s `index` method and edit the call that loads the list of players. We'll use the web session instead of the database session to try loading the `Player` objects.

Listing 15-27.

```
def index(self):
    db_session = meta.Session()
    c.page_title = 'Player List'
    c.players = session.query(Player).all()
    return render('list_players.html')
```

Load <http://localhost:5000/> to see the error page.

There's a lot of information that Pylons throws back at you. Along the top of the screen, you'll see four tabs: Traceback, Extra Data, Template, and Source. Pylons will have put you in the Traceback tab by default to start with. If you look at the error, you'll see the exact line number in the source file that the error occurred in. What's special about Pylons' traceback tab is that this is actually a fully interactive session.

You can select the + signs to expand each stack frame and a text input along with some local variables on that frame will be revealed. That text input is an interface into your server process. You can type virtually any Python command into it, hit Enter, and you will get back live results. From here, we can see that we should have used `db_session` and not the `session` variable. See Figure 15-2.

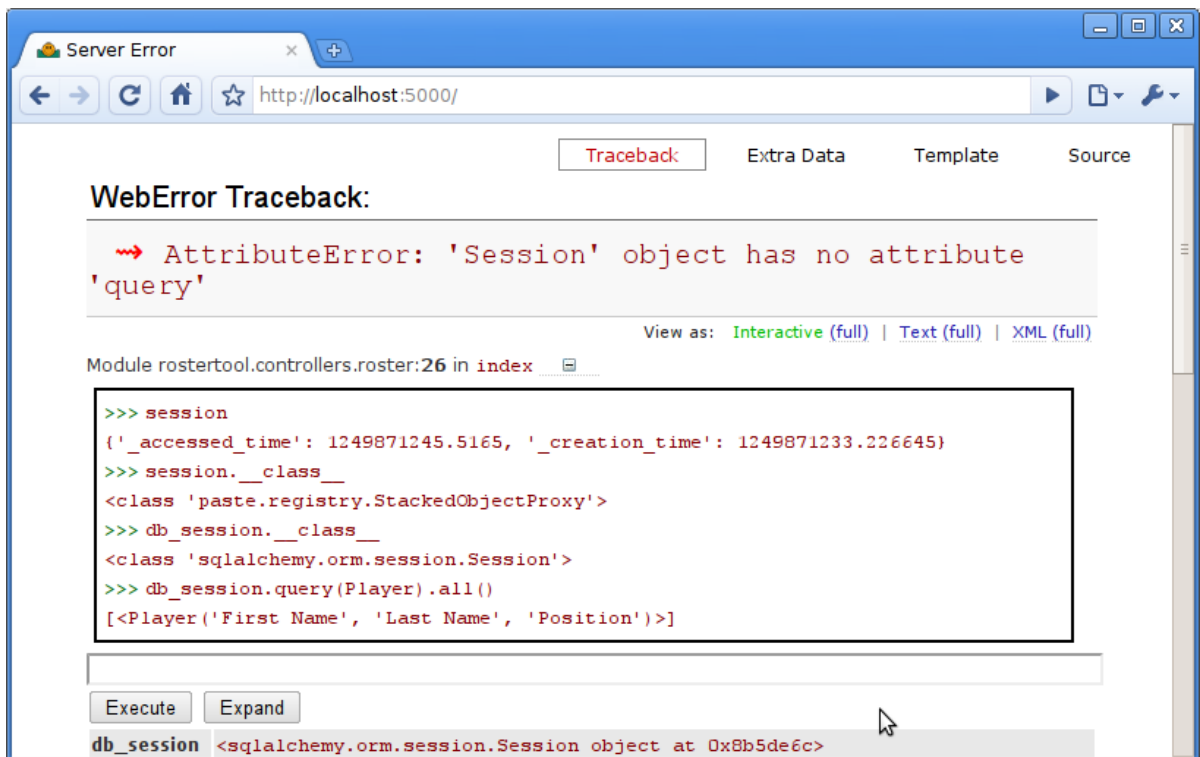


Figure 15-2. Error message caused by use of `Session`

This is pretty fantastic. If you click on the View link, you can even jump to the full source listing of the Python module that caused the error. One bug in Pylons at the time of writing is that sometimes the hyperlink is malformed. So, although the traceback will correctly list the line number that the error occurred at, the source listing may go to the wrong line.

The Pylons developers have also embedded an interface into search engines to see if your error has been previously reported. If you scroll down to the bottom of

your traceback page, you'll see another tab control with a Search MailLists option. Here, Pylons will automatically extract the exception message and provide you an interface so you can literally search all the mailing lists that are relevant to your particular Pylons installation.

If you can't find your error on the mailing lists, you can go to the next tab, Post Traceback, and submit your stacktrace to a webservice on PylonsHQ.com so that you can try to debug your problems online with other collaborators. Combining unit tests, functional tests, and the myriad of debugging options afforded to you in the web debugger, Pylons makes the debugging experience as painless as possible.

Deployment into a Servlet Container

Deploying your pylons application into a servlet container is very straight forward. Just install snakefight from PyPI using easy_install and you can start building WAR files.

Listing 15-28.

```
$ easy_install snakefight
...snakefight will download and install here ...
$ jython setup.py bdist_war --paste-config test.ini
```

By default, snakefight will bundle a complete instance of your Jython installation into the WAR file. What it doesn't include is any JAR files that your application depends on. For our small example, this is just the postgresql JDBC driver. You can use the `--include-jars` options and provide a comma separated list of JAR files.

Listing 15-29.

```
$ jython setup.py bdist_war \
  --include-jars=postgresql-8.3-604.jdbc4.jar \
  --paste-config=test.ini
```

The final WAR file will be located under the dist directory. It will contain your postgresql JDBC driver, a complete installation of Jython including anything located in site-packages and your application. Your war file should deploy without any issues into any standards compliant servlet container.

Summary

We've only scratched the surface of what's possible, but I hope you've gotten a taste of what is possible with Pylons. Pylons uses a large number of packages so you will need to spend more time getting over the initial learning curve, but the dividend is the ability to pick and choose the libraries that best solve your particular problems. It would be helpful

to take a look at some other resources such as *The Definitive Guide to Pylons* from Apress, which is also available online at <http://pylonsbook.com>.

Source: <http://www.jython.org/jythonbook/en/1.0/IntroToPylons.html>