CHAPTER 3

Exploring Pylons

Now that you have seen how to set up a Python environment and have used Easy Install to install Pylons and its dependencies, it is time to create your first Pylons project. In time-honored tradition, I'll start with an example that outputs the words `Hello world!` to the web browser.

In this chapter, you'll also learn the basics of the Hypertext Transfer Protocol, learn about Pylons' request and response objects, and begin learning how Pylons works and about the other objects Pylons sets up for you.

Exploring Pylons' Dependencies

As you learned in Chapter 1, Pylons is more like a collection of very carefully chosen separate components than a tightly integrated framework, but it can often be difficult to work out exactly how each of the components fits together. You'll learn this over the course of the book, but let's take a quick look at the packages that were installed along with Pylons so that you can get an idea of the big picture.

If you followed the installation instructions from the previous chapter, take a look at the `lib/python2.5/site-packages` directory in your virtual Python environment. You will see all of Pylons' dependencies as well as the `easy-install.pth` and `setuptools.pth` files, which Easy Install uses to keep track of the installed eggs. If you've installed other packages too, you will also see them.

The following list contains the components relevant to Pylons 0.9.7. Of course, the version numbers might change slightly over time and future versions of Pylons might have slightly different dependencies, but the following list is correct at the time of this writing:

`Beaker-0.9.5-py2.5.egg`: Beaker is a piece of software used internally by Pylons to implement its caching and session functionality. The Pylons `session` global described later in the chapter uses Beaker, as does Pylons' caching functionality described in the Pylons Cookbook at `http://wiki.pylonshq.com/display/pylonsdocs/Caching+in+Templates+and+Controllers`, but you would never normally interact with Beaker yourself directly.

`decorator-2.2.0-py2.5.egg`: This is a simple tool used by Pylons to create the `@validate` and `@jsonify` decorators. You'll learn about `@validate` in Chapter 6, and you'll learn about `@jsonify` in Chapter 15. Once again, you won't normally use `decorator` in your own programs because you'll usually use the decorators provided by Pylons.

`FormEncode-1.0.1-py2.5.egg`: FormEncode is a library for validating form submissions from web sites. Although Pylons doesn't use it internally, Pylons users work with it so often that it is considered an essential part of Pylons. The FormEncode package also includes a module named `formencode.htmlfill` that can be used to populate a string containing HTML fields with values and error messages. Together FormEncode and HTML Fill make an ideal tool set for handling forms in a Pylons application. Chapter 6 is dedicated to explaining how to use FormEncode and HTML Fill in a Pylons application.

`Mako-0.2.0-py2.5.egg`: Mako is one of the three template languages that Pylons 0.9.7 supports out of the box. The others are Genshi (an XML template language) and Jinja (based on Django's template system). You have to install Genshi and Jinja separately if you want to use them, whereas Mako is included in the default Pylons installation because it is the recommended template language to use. Using Mako to generate your views is described in detail in Chapter 5.

`nose-0.10.3-py2.5.egg`: This provides tools to help you write and run automated unit tests. Testing is described in Chapter 12.

`Paste-1.6-py2.5.egg`, `PasteDeploy-1.3.2-py2.5.egg`, and `PasteScript-1.6.3-py2.5.egg`: Paste comes in three packages for the benefit of framework developers who require only one part of its functionality. Pylons uses all three packages for a wide variety of things throughout the framework, but once again, as a Pylons application developer, you won't normally directly interact with the Paste components yourself.

Over time, the functionality in the Paste modules has been split up into custom packages. For example, the `paste.wsgiwrappers` module, which provided the `pylons.request` and `pylons.response` objects in Pylons 0.9.6, is now replaced by WebOb, which provides the Pylons 0.9.7 versions of those Pylons objects. The `paste.eval_exception` module, which provided the 0.9.6 error handling, is replaced by WebError in Pylons 0.9.7, and even the `paste.auth` functionality has been built upon and improved in AuthKit, which you'll learn about in Chapter 18. Don't be surprised if future versions of Pylons include even more projects spun out from their roots in Paste.

Despite the gradual shift to separate packages, Pylons still relies on Paste for its configuration files, registry manager, development HTTP server, project template creation, test fixtures, error documents, and more. The various parts of Paste are described throughout the book as they are encountered.

`Pylons-0.9.7-py2.5.egg`: This is where everything needed to glue together the other components of Pylons is found. Pylons itself is relatively small, so if you are the curious type, feel free to look at its code to get a feel for how everything works.

`Routes-1.9-py2.5.egg`: Pylons uses a system called Routes that allows you to map a URL to a set of variables usually including `controller` and `action`. These variables are then used to determine which Pylons controller class and method should be used to handle the request. At the same time, Routes allows you to specify a set of variables and have a URL generated from them so that you never need to hard-code URLs into your application. I'll introduce Routes in this chapter, but you will learn the details of all of Route's powerful features in Chapter 9.

`setuptools-0.6c8-py2.5.egg`: This contains the methods used by the `easy_install` script to provide all of its features and allow the use of egg files.

`simplejson-1.8.1-py2.5-linux-x86_64.egg`: This package converts data back and forth between JSON and Python formats and is used by the `@jsonify` decorator mentioned earlier. Pylons application developers also occasionally use `simplejson` directly in their controllers.

`Tempita-0.2-py2.5.egg`: Tempita is a small template language that is a dependency of Paste. It is used only behind the scenes for simple variable substitutions when you create a new Pylons project directory with the `paster create` command described later in this chapter.

`WebError-0.8-py2.5.egg`: WebError provides Pylons' powerful interactive debugging and traceback functionality described in Chapter 4.

`WebHelpers-0.6-py2.5.egg`: WebHelpers is a collection of stand-alone functions and classes that provide useful functionality such as generating common HTML tags and form fields, handling multiple pages of results, and doing much more.

`WebOb-0.9.2-py2.5.egg`: This provides the new `pylons.request` and `pylons.response` objects in Pylons 0.9.7.

You might have noticed that SQLAlchemy, a database toolkit you'll learn about in Chapter 7, and AuthKit, a toolkit you'll learn about in Chapter 18, are not included in the list of packages installed automatically with Pylons. This is because Pylons can be used perfectly well without them, and although most users will choose to install them, some developers will want to choose alternatives instead.

Installing Pylons also installs some scripts. If you look in your virtual Python environment's `bin` directory (or the `Scripts` directory on Windows), you will see the following:

`activate` (or `activate.bat` on Windows): This is an optional script described in Chapter 2 for activating a virtual Python environment to make the other scripts available automatically on the current shell or command prompt without having to type the full path to the virtual Python environment.

`nosetests`: This is a script used for running your Pylons tests; it is provided by the `nose` package, which was mentioned earlier and will be described in Chapter 12.

`python`: This is the Python executable you should use for all work within the virtual Python environment.

`easy_install`: This is the Easy Install program for installing software into your virtual environment described in Chapter 2. `mako-render`: This is a simple script installed with Mako that takes a single file containing Mako template markup as an argument and outputs the rendered template to the standard output on your console. This isn't very useful for Pylons development.

`paster`: This is a very useful script that uses the Paste Script package and has a number of subcommands including `paster create` and `paster serve`, which you'll see later in this chapter, that are for creating a new Pylons project and serving a Pylons application, respectively. You'll also see `paster make-config` and `paster setup-app`, which are for handling the creation of a config file from a distributed Pylons project and for setting it up. These are advanced features you'll learn about in the SimpleSite tutorial throughout the book.

Your `bin` (or `Scripts`) directory might also see a file such as `easy_install-2.5`, which is simply a Python 2.5 version of the `easy_install` script, or a `python2.5` script, which is a Python 2.5 version of the `python` script if you are using multiple versions of Python on your system. You should generally use the `easy_install` and `python` versions because they match the version of Python you used when you ran the `python virtual_python.py env` command in Chapter 2.

Don't worry if you don't understand everything I've mentioned in this section; it will all become clear as you get more familiar with Pylons.

Creating a Pylons Project

Now that you've seen a quick overview of all the Pylons components, it's time to create your first Pylons project.

Let's get started by using Paste's `paster create` command to automatically generate a Pylons project directory structure for you. You are completely free to create all the files and directories yourself if you prefer, but the Pylons project template provides a useful starting point that most people prefer to use.

■Caution In Pylons terminology, there are two different types of template, and it is important not to get confused between the two. *View templates* are text files that contain a mixture of Python and HTML and are used to generate HTML fragments to return to the browser as the view component of the MVC architecture. Any template written with Mako is a view template. *Project templates* are sets of files and directories that are used by the `paster create` command to generate a complete project directory structure to use as a basis for a new Pylons application.

Create a new project named `HelloWorld` based on the Pylons default project template with this command:

```
$ paster create --template=pylons HelloWorld
```

The `--template` option tells the `paster create` command which project template to use to create the `HelloWorld` project. You will also see examples using `-t` instead of `--template`, but both have the same effect:

```
$ paster create -t pylons HelloWorld
```

■Note If you have problems running the previous `paster create` command, it is likely you have forgotten to include the full path to the `paster` script (or the `paster.exe` executable in the case of Windows). You might see a message such as `paster: command not found` or `'paster' is not recognized as an internal or external command, operable program or batch file`.

Remember that if you are using a virtual Python environment, you will need to type the full path to the executable (for example, `env/bin/paster` or `C:\env\Scritps\paster`) or modify your `PATH` as described in Chapter 2, either directly or by using the `activate` or `activate.bat` script. The examples in this book assume you have read Chapter 2 and will type whatever is appropriate for your installation.

The project template used in the previous `paster create` command is called `pylons`, which is the default for Pylons. You can always see which project templates are available with the `--list-templates` option shown here, but bear in mind that because Paste is a general-purpose library, not all the templates available will be for generating Pylons projects. In Chapter 19, you'll learn how to create your own Pylons project template to add to this list:

```
$ paster create --list-templates
Available templates:
  basic_package:   A basic setuptools-enabled package
  paste_deploy:    A web application deployed through paste.deploy
  pylons:          Pylons application template
  pylons_minimal:  Pylons minimal application template
```

Advanced users might like to experiment with the `pylons_minimal` application template, which leaves you to do more configuration yourself.

Now let's return to the `HelloWorld` example. Once you have run the `paster create --template=pylons HelloWorld` command, you will be asked some questions about how you want your project set up. You can choose which view template language you want to use for the project and whether you want SQLAlchemy support built in. For this example, choose the defaults of Mako for the view template language and no SQLAlchemy support since you haven't installed SQLAlchemy yet anyway. Just press Enter at each of the prompts, and the script will quickly generate your new project.

Serving a Pylons Application

Now that you have a sample project application, it is a good idea to run it with a web server to see what the default project provides; however, before you can serve the Pylons application you've just created, you need to learn about configuration files.

Configuration Files

Configuration files enable you to set up the same Pylons application on different servers with different settings and for different purposes without having to change the source code for the project.

The project template you have used creates two configuration files for you in the `HelloWorld` directory, one named `development.ini` and one named `test.ini`. The `development.ini` file contains settings to run the Pylons application in a development environment such as your local workstation, whereas the `test.ini` file contains the settings you want to use when testing the application. You create a new configuration file called `production.ini` when you are ready to serve the application file in production.

The following code is the top part of the `development.ini` file generated as part of the application template. There is also some logging configuration, which you'll learn about in Chapter 20.

```
#
# helloworld - Pylons development environment configuration
#
# The %(here)s variable will be replaced with the parent directory of this file
#
[DEFAULT]
debug = true
# Uncomment and replace with the address which should receive any error reports
#email_to = you@yourdomain.com
smtp_server = localhost
error_email_from = paste@localhost

[server:main]
use = egg:Paste#http
host = 127.0.0.1
port = 5000

[app:main]
use = egg:helloworld
full_stack = true
cache_dir = %(here)s/data
beaker.session.key = helloworld
beaker.session.secret = somesecret
# If you'd like to fine-tune the individual locations of the cache data dirs
# for the Cache data, or the Session saves, un-comment the desired settings
# here:
#beaker.cache.data_dir = %(here)s/data/cache
#beaker.session.data_dir = %(here)s/data/sessions

# WARNING: *THE LINE BELOW MUST BE UNCOMMENTED ON A PRODUCTION ENVIRONMENT*
# Debug mode will enable the interactive debugging tool, allowing ANYONE to
# execute malicious code after an exception is raised.
#set debug = false
```

As you can see, the configuration file is in three parts. The [DEFAULT] section contains global configuration options that can be overridden in other sections. The [server:main] part contains information for the server used to serve the Pylons application, and the [app:main] section contains configuration options for the Pylons application. All the option values can contain the string %(here)s, which gets replaced with the location of the config file, enabling you to easily specify relative paths.

Let's discuss the options available:

`debug`: This can be `true` or `false`. If `true`, the Pylons interactive debugger is enabled to allow you to track down problems, as you'll learn about in the next chapter. You should always disable the interactive debugger in production environments by setting `debug` to `false`.

`email_to`, `smtp_server`, `error_email_from`: These options specify where error reports should be sent if the interactive debugger is disabled, but they could potentially be used by third-party components too since they are specified in the `[DEFAULT]` section and are therefore available to all components using the configuration file.

`host`, `port`: These specify the IP address and port the server should listen on for requests. Only the server uses them.

`full_stack`: You can set this to `false` to disable Pylons interactive debugging, error report support, and error documents support, but you'll usually leave this set to `true`.

`cache_dir`: This is the directory where components can store information on the filesystem. Session information from Beaker and cached view templates from Mako are stored in this directory. You can manually override where Beaker stores its session information with the `beaker.cache.data_dir` and `beaker.session.data_dir` options, but you won't usually need to do so.

`beaker.session.key`: This should be something unique to your application so that other applications using Beaker can't access your application's data.

`beaker.session.secret`: This is a random string of letters and numbers used to make the cookie value representing the session harder to guess to reduce the risk of a malicious user trying to gain access to someone else's session information. You should always change this to something other than `somesecret` if you are using Pylons session functionality.

The Paste HTTP Server

To run your application for development purposes, it is recommended you use the Paste HTTP server from the Paste package that was installed as one of Pylons' dependencies. The Paste HTTP server does for Pylons applications what Apache does for PHP and other languages; it listens for HTTP requests and dispatches them to the running application, returning the result via HTTP to the user's browser.

The Paste HTTP server has two features that make it more suitable for Pylons development than most web servers:

- It can be made to automatically reload when you change the source code.
- It understands the configuration files used by Pylons, so they can be used directly. As you'll see in Chapter 19, other servers require the assistance of the Paste Deploy package to turn a Pylons config file into an application.

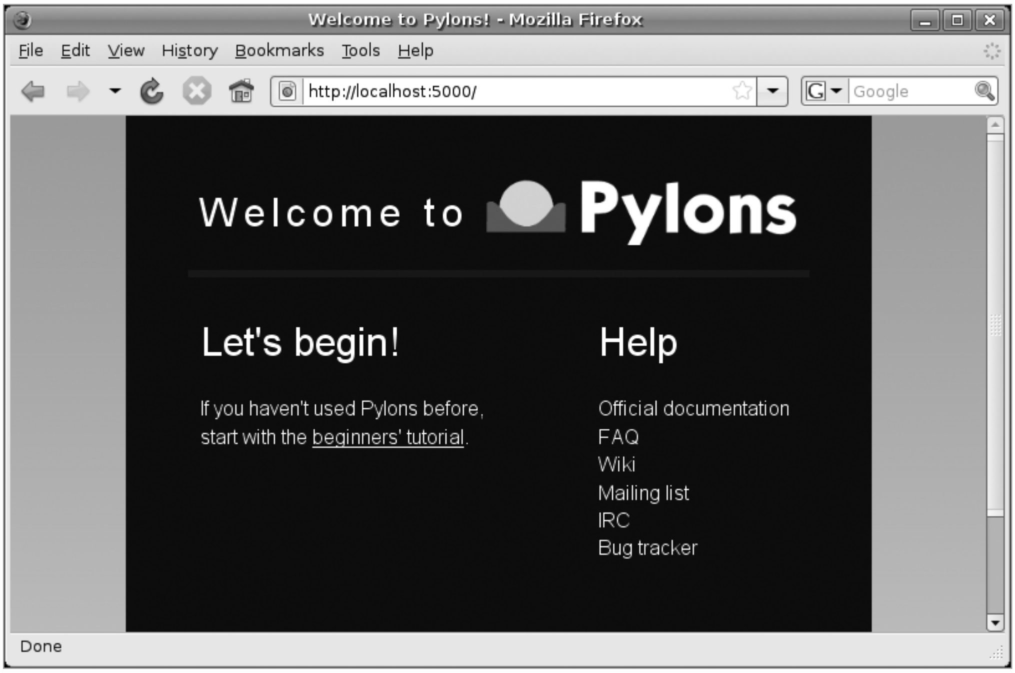
You can start the server with your development configuration with this command:

```
$ cd HelloWorld
$ paster serve --reload development.ini
Starting subprocess with file monitor
Starting server in PID 17586.
serving on 127.0.0.1:5000 view at http://127.0.0.1:5000
```

If you are Windows user, you may be prompted to unblock Python on port 5000 depending on your firewall settings.

The `--reload` option puts the Paste HTTP server into a very useful mode where the server carefully monitors all Python modules used by your application as well as the `development.ini` configuration file. If any of them change, the server is automatically reloaded so that you can immediately test your changes. This is useful during development because it saves you from having to manually stop and start the server every time you make a change.

To stop the server, you can press Ctrl+C (or Ctrl+D if you're running Windows), but don't stop it yet. If you visit `http://127.0.0.1:5000/` in your web browser when the server is running, you will see the welcome page shown in Figure 3-1.


Welcome to Pylons! - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:5000/
Google
Done

Welcome to Pylons

Let's begin!

If you haven't used Pylons before, start with the beginners' tutorial.

Help

Official documentation
FAQ
Wiki
Mailing list
IRC
Bug tracker

Figure 3-1. *The Pylons default page*

Static Files

Now that the server is running, try creating a new file named `hello.html` in the `HelloWorld/helloworld/public` directory with the following content:

```
<html>
    <body>
        Hello world!
    </body>
</html>
```

If you visit `http://127.0.0.1:5000/hello.html`, you will see the `Hello world!` message.

A Pylons project's `public` folder is a bit like an `htdocs` directory in Apache. Any files in the `public` directory are treated as static files and are served treating the URL as the path to the file to serve. If the URL represents a directory and that directory contains an `index.html` file, it will be served instead; however, for security reasons, Pylons does not provide a directory index facility, so if no `index.html` file is present in a directory, a "404 File Not Found" response is returned.

Pylons automatically provides E-Tag caching for static files in the `public` directory. This allows browsers that support E-Tag caching to check to see whether a file has changed since the last time they fetched it. If it hasn't changed, the browser won't download the file a second time.

A Word About IP Addresses, Hosts, and Security

The numbers `127.0.0.1` in the URL you have been using to test your `HelloWorld` application represent the IP address on which the Paste HTTP server is serving the Pylons application. The IP address `127.0.0.1` is a special IP address that references your own computer. This means your Pylons application can be accessed only from the computer on which you are running the Paste HTTP server. This is actually very important because, as you will learn in the next chapter, Pylons comes with a powerful interactive debugging tool that is enabled by default when using the `development.ini` configuration file. If people could access your running development instance and an error occurred, they might be able to use the interactive debugger to enter malicious commands, so this is why as of Pylons 0.9.7 the default configuration file instructs the Paste HTTP server to bind to `127.0.0.1` so it answers requests only from the local computer.

If you want to test how the application works from a different computer, you can change the IP address the Paste HTTP server uses by editing the `host` option in the `[server:main]` section of the `development.ini` file. You can also change the port on which the server runs by changing the `port` variable. If you don't want `:5000` in your URLs, you should set the server to run on port 80. Browsers will connect to port 80 by default for HTTP requests if no port is specified in the URL. Most production systems run on port 80 for this reason, but for development it is fine to run your application on port 5000.

The IP address `0.0.0.0` is also worth knowing about. Setting the `host` option to `0.0.0.0` will cause the Paste HTTP server to respond to requests on all IP addresses on your server. This is the setting normally used in production configurations when the Pylons interactive debugger has been disabled.

Servers frequently have hostnames mapped to particular IP addresses, and you can specify a server's hostname instead of the IP address if you prefer. It is a convention on most platforms that the hostname localhost will refer to the IP address `127.0.0.1`, so for the majority of the time, you can use localhost rather than the IP address `127.0.0.1`. With the Paste HTTP server still running, try visiting `http://localhost:5000/hello.html`; chances are you will still see the same `Hello World!` message you did before.

Exploring a Pylons Project's Directory Structure

Now that you've seen a simple Pylons application serving static files from the `public` directory, I'll take the opportunity to show you the rest of the files and directories that the `paster create` command generated for you from the default `pylons` application template.

The main `HelloWorld` directory contains the following files and directories:

`docs`: This directory is where you can keep documentation for your project. Pylons applications are often documented in a language called reStructuredText. The reStructuredText files can then be converted to HTML using tools such as Sphinx. Both reStructuredText and Sphinx are discussed in Chapter 13.

`helloworld`: This is the main application directory, but its name depends on the package name you gave as the argument to the `paster create` command when the project was generated. Pylons applications are usually given a package name in CamelCase, but the application directory itself is the lowercase version of the package name. In this case, you specified the package name as `HelloWorld`, so the main Pylons application directory is named `helloworld`. If you were to write `import helloworld`, it would be this directory's files that are imported. I'll return to this directory in a moment to explore the subdirectories it contains.

`HelloWorld.egg-info`: This is a special directory that contains metadata about your project in a format that is used by `setuptools` when you treat the application as an egg.

`development.ini` and `test.ini`: These are the configuration files I discussed in the previous section.

`ez_setup.py`: Your Pylons application relies on some features provided by the `setuptools` module, but not every Python installation comes with the `setuptools` module already installed, because it isn't an official part of the Python distribution. The `ez_setup.py` file is therefore included to automatically install `setuptools` if someone without it tries to install your Pylons application.

`MANIFEST.in`: Your Pylons application contains various files that aren't Python modules such as the templates and static files. These files are included in a Python package only if they are specified in a `MANIFEST.in` file. The `MANIFEST.in` file in your project's directory forces these files to be included.

`README.txt`: Having a `README` file in a project's root directory is standard practice, so this file is simply there for you to describe your project to anyone looking at its source code. You can customize it as you see fit.

`setup.cfg` and `setup.py`: The `setup.py` and `setup.cfg` files control various aspects of how your Pylons application is packaged when you distribute it. They also contain metadata about the project. You'll see them being used as you read the SimpleSite tutorial chapters.

You may also notice a `data` directory that contains cached session and template information. It is created the first time you run a Pylons application that needs it to be present. The location of this directory can be configured with the `cache_dir` option you saw a moment ago when I discussed configuration file options.

Now let's take a look at the main Pylons application directory. As mentioned a moment ago, this will have a name based on the package name, so it will be different in each Pylons project. In this case, it is called `helloworld` and contains the following files and directories:

`config`: The `config` directory is where most Pylons functionality is exposed to your application for you to customize.

`controllers`: The `controllers` directory is where your application controllers are written. Controllers are the core of your application. They allow you to handle requests, load or save data from your model, and pass information to your view templates for rendering; they are also responsible for returning information to the browser. You'll create your first controller in the next section.

`lib`: The `lib` directory is where you can put Python code that is used between different controllers, third-party code, or any other code that doesn't fit in well elsewhere.

`model`: The `model` directory is for your model objects; if you're using an object-relational mapper such as SQLAlchemy, this is where your tables, classes, and relations should be defined. You'll look at using SQLAlchemy as a model in Chapter 7.

`public`: You've already seen the `public` directory. It is similar to the `htdocs` directory in Apache and is where you put all your HTML, images, JavaScript, CSS, and other static files.

`templates`: The `templates` directory is where view templates are stored.

`tests`: The `tests` directory is where you can put automated unit tests for your application.

`__init__.py`: The `__init__.py` file is present so that the `helloworld` directory can be imported as a Python module within the egg.

`websetup.py`: The `websetup.py` contains any code that should be executed when an end user has installed your Pylons application and needs to initialize it. It frequently contains code to create the database tables required by your application, for example. We'll discuss this in Chapter 8.

Creating a Controller and Modifying the Routes

It's now time to learn how to generate the message dynamically using a Pylons *controller*. Controllers are the basic building blocks of Pylons applications. They contain all the programming logic and can be thought of as mini-applications. Controllers are implemented as Python classes. Each method of the class is known in Pylons as an *action*. On each request Pylons routes the HTTP information to a particular controller action based on the URL that was requested. The action should return a response, which Pylons passes back to the server and on to the browser.

Let's see this in practice by creating a controller. Once again you can use a `paster` command to help you get started quickly:

```
$ paster controller hello
```

This command creates a skeleton `controllers/hello.py` file for you as well as a `helloworld/tests/functional/test_hello.py` file that is used for running some of the functional tests of the controller discussed in Chapter 12. If you are using the Subversion revision control system to manage the source files in a Pylons project, the `paster controller` command will also automatically add both files to your working copy.

Modify the `index()` action of the `HelloController` to look like this:

```
class HelloController(BaseController):
    def index(self):
        return 'Hello from the index() action!'
```

Let's test this code. With the server still running, visit `http://127.0.0.1:5000/hello/index`, and you should be greeted with the `Hello from the index() action!` message.

Let's look closely at that URL. If you have used ASP, PHP, or CGI scripts, you'll know that the URL the user visits directly represents the path on the filesystem of the script the server should execute. Pylons is much more flexible than this; it uses a system called Routes to map sets of URLs to particular controllers. This means your URLs don't always have to directly represent the controllers that handle them. By default, Routes is set up so that the first URL fragment after the hostname and port represents the controller and the second part represents the action. In this case, /hello/index means use the index action of the hello controller, and so the URL you just visited results in the index() method of HelloController being called to display the response.

One very common requirement is the ability to map the root URL http://localhost:5000/ to a controller action. After all, you wouldn't want to be limited to having a static file as the root URL. To do this, you need to add a line to the routes configuration. Add a new line to the top of the main route map in helloworld/config/routing.py just after the # CUSTOM ROUTES HERE comment so that the routes are defined like this:

```
# CUSTOM ROUTES HERE

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

This tells Routes that the root URL / should be mapped to the index action of HelloController. Otherwise, Routes should look for URLs in the form /controller/action/, and /controller/action/id. This, along with other details of Routes, is described in detail in Chapter 9.

Since you have made changes to a Python file used by the project, you would need to restart the server for the changes to be picked up. Because you started the server with the --reload option, though, this will happen automatically.

If you visit http://127.0.0.1:5000/ again, you will notice that the static welcome page is still there. This is because Pylons looks in the public directory for files to serve before attempting to match a controller. Because the public/index.html file still exists, Pylons serves that file.

If you delete the public/index.html file, you should see the Hello from the index() action! message you were expecting.

Understanding How HTTP Works

At this point it is worth taking a step back from Pylons to understand what is actually going on to display the Hello from the index() action! message.

At its heart, web development is all about the Hypertext Transfer Protocol (HTTP). Any web page you visit that starts with http:// is using HTTP to communicate between the browser and the server. Other protocols are used on the Internet too, such as the File Transfer Protocol (FTP), but for creating data-driven web sites with Pylons, HTTP is the only one you need to understand.

■ **Note** If you're new to web development, it is important not to get confused between HTTP and HTML. HTTP is the protocol with which the browser communicates with a server, whereas HTML is a markup language used to create web pages.

To understand exactly what is going on when you create a web page, it is useful to be able to see the HTTP information being sent back and forth between a web browser and a Pylons application. One good tool for doing this is the LiveHTTPHeaders extension for the Firefox web browser, which you can download from http://livehttpheaders.mozdev.org/. Once you have installed it, you can select View ➤ Sidebar ➤ LiveHTTPHeaders from the menu to load the extension in the sidebar, and it will display all the HTTP information sent and received on each request.

Tip You can download the Firefox web browser from `http://mozilla.com/products/firefox`. It will run on the Windows, Mac OS X, Linux, and BSD platforms. It is particularly useful for web development because of the extensions available that give you fuller access to the processes going on within the web browser.

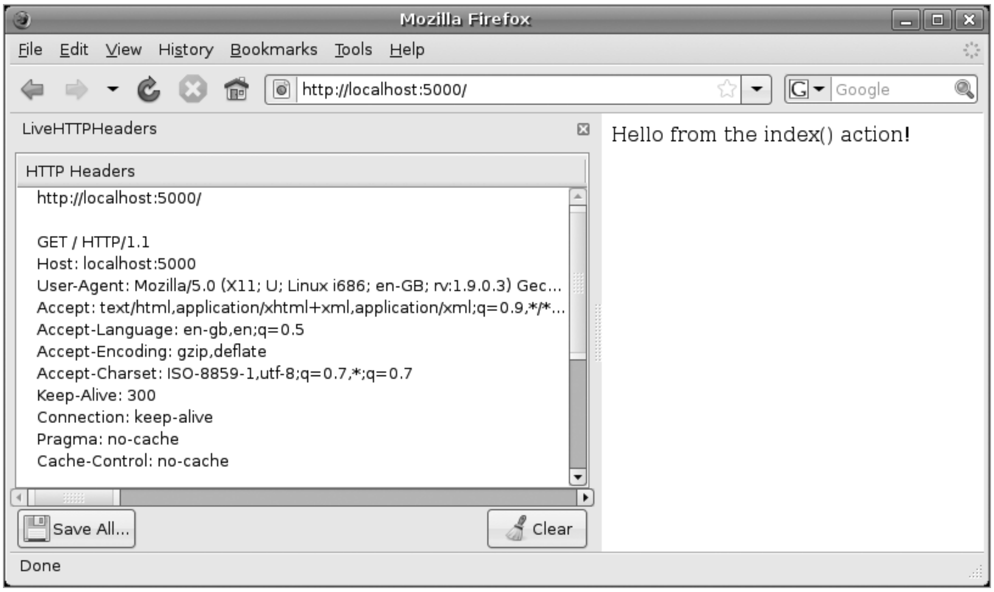
Another particularly useful extension is the Web Developer toolbar available from `https://addons.mozilla.org/en-US/firefox/addon/60`, which offers facilities for managing cookies and style sheets as well as for outlining block-level elements and tables.

Firefox also has powerful extensions such as Firebug, which in addition to its JavaScript and DOM manipulation facilities allows you to analyze page load times. I will discuss Firebug in Chapter 15 when I cover Ajax.

When you request a page, the browser sends an HTTP request to the server. When the server receives that request, it will calculate an HTTP response. Depending on the request, it may retrieve information from a database or read a file from the filesystem to prepare the response.

HTTP supports different types of requests. You are probably already familiar with the GET method used to retrieve information from a URL and the POST method used primarily to send form data, but there are other less well-known methods such as HEAD, OPTIONS, PUT, DELETE, TRACE, and CONNECT.

Figure 3-2 shows a simple HTTP GET request where you can see the HTTP information sent when visiting `http://127.0.0.1:5000/`.



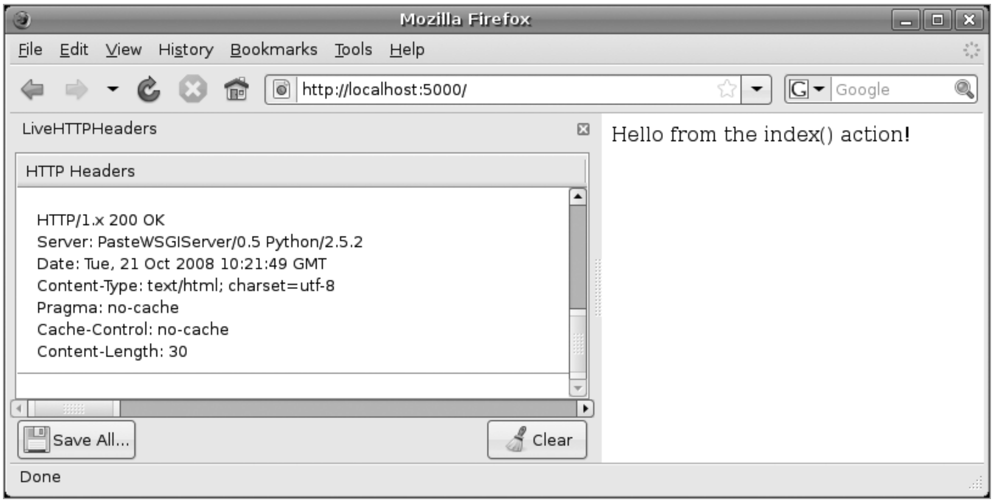
```
http://localhost:5000/

GET / HTTP/1.1
Host: localhost:5000
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-GB; rv:1.9.0.3) Gec...
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*...
Accept-Language: en-gb,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Pragma: no-cache
Cache-Control: no-cache
```

Hello from the index() action!

Figure 3-2. *An HTTP request in LiveHTTPHeaders*

Figure 3-3 shows the response returned.



```
HTTP/1.x 200 OK
Server: PasteWSGIServer/0.5 Python/2.5.2
Date: Tue, 21 Oct 2008 10:21:49 GMT
Content-Type: text/html; charset=utf-8
Pragma: no-cache
Cache-Control: no-cache
Content-Length: 30
```

Hello from the index() action!

Figure 3-3. *An HTTP response in LiveHTTPHeaders*

As you can see, the browser sends quite a lot of information to the server. The application then processes this information and performs any necessary operations before returning a status and any HTTP headers it wants to send back. Of particular note is the `Content-type` header that tells the browser what sort of content is going to be sent (in this case `text/html`). Finally, the application returns the content that will be displayed by the browser (in this case the HTML that makes up the page). The server may add extra HTTP headers or perform other modifications before the response is returned.

In this example, the HTTP status code is 200, which means everything went fine and no error occurred. The application and server can use many other status codes to tell the browser what happened while processing the request. Table 3-1 describes some of the most commonly used codes.

Table 3-1. *Commonly Used HTTP Status Codes*

Status Code	Description
200 OK	The request has succeeded.
401 Unauthorized	The request requires user authentication.
403 Forbidden	The server won't let the user access what was requested, perhaps because the user doesn't have the appropriate permissions.
404 Not Found	The server has not found anything matching the request URI.
500 Internal Server Error	The server encountered an unexpected condition that prevented it from fulfilling the request.

For the vast majority of web development situations, the areas of the protocol I have discussed are all you need to know, but if you are interested in the details of HTTP, you can find the full specification at `http://www.w3.org/Protocols/rfc2616/rfc2616.txt`, including an explanation of all the request methods and status codes.

Exploring the Environment

Information about the HTTP request as well other information is presented to your Pylons application through environment variables. These are a set of dynamic values set by the web server on each request. Exactly which variables are set depends on the web server, the browser, and the action the user is trying to perform. Once Pylons receives these variables from the environment, they are passed to the Pylons `request` object for use in your controllers. You wouldn't usually access environment variables directly, but it is useful to know where the Pylons `request` object gets its information from.

If you have ever written CGI scripts or used PHP, you will most likely be familiar with using environment variables already, in which case many of the variables mentioned in this section will be familiar. It is worth remembering, though, that Pylons controllers aren't executed in the same way as CGI or PHP scripts, so consequently some of these variables have slightly different meanings in the Pylons context.

The following variable is set for all requests and are not request specific:

`SERVER_NAME`: The server's hostname, DNS alias, or IP address as it would appear in self-referencing URLs.

The following are specific to the request:

`SERVER_PORT`: This is the number of the port to which the request was sent.

`SERVER_PROTOCOL`: This is the name and revision of the protocol used in the request. Usually it is `HTTP/1.1`.

`REQUEST_METHOD`: This is the method with which the request was made, such as GET, HEAD, POST, and so on.

`REMOTE_HOST`: This is the hostname making the request (set only if the server has this information).

`REMOTE_ADDR`: This is the IP address of the remote host making the request.

`REMOTE_USER`: This is the username of an authenticated user but is set only if a user is signed in.

`AUTH_TYPE`: If the server supports user authentication, this is the authentication method used to validate the user.

`CONTENT_TYPE`: If the request was an HTTP POST or PUT, then there could be content sent by the client. This is the content type of the data.

`CONTENT_LENGTH`: If a `CONTENT_TYPE` is specified, this is the length of that content.

`QUERY_STRING`: This is the part of the URL after a `?`, such as `foo1=bar1&foo2=bar2`.

`SCRIPT_NAME` and `PATH_INFO`: In a Pylons application, `SCRIPT_NAME` is the part of the URL before the URL the Pylons application would treat as its site root, and `PATH_INFO` is the part of the URL after it. If you are serving the application, normally `SCRIPT_NAME` will be `''` and `PATH_INFO` will be the part of the URL before the query string. If you were to mount the Pylons application at, say, `/myapp`, then `SCRIPT_NAME` would be `/myapp` and `PATH_INFO` would be the part of the URL after it and before the query string.

In addition to these environment variables, any HTTP headers that are sent in the request and not dealt with by previous environment variables are also included after being prefixed with `HTTP_` and having any `-` characters replaced with `_` characters. Since these are set by the user's browser, they warrant more suspicion than the previous environment variables. Here are some of the more familiar ones as examples:

`HTTP_HOST`: This is the hostname and domain name portion of the URL, such as `www.pylonshq.com`.

`HTTP_COOKIE`: This is the content of the client's cookie(s).

`HTTP_USER_AGENT`: This is the user-agent string to identify the browser type and version.

Although you would normally access these variables through the more convenient `request` object, Pylons is all about giving power to the developer, so you can still access environment variables directly in your controllers if you choose. They are available as the `request.environ` dictionary. You'll see how to use the `request` object in a few moments.

In addition to the CGI-style variables listed earlier, the server running your Pylons application also adds extra information to the environment called *WSGI variables*, which can sometimes be useful to know about for use in your Pylons application.

■ **Tip** WSGI stands for the Web Server Gateway Interface, and although you don't need to know anything about it to develop Pylons applications, it is actually a very powerful API on which much of Pylons is based, so I'll cover it in detail in Chapters 16 and 17.

Here are the WSGI variables you will also find in the Pylons `request.environ` dictionary:

`wsgi.version`: This is a tuple, `(1,0)`, representing WSGI version 1.0.

`wsgi.url_scheme`: This is a string representing the "scheme" portion of the URL at which the application is being invoked. Normally, this will have the value `http` or `https`, as appropriate.

`wsgi.input`: This is an input stream (a file-like object) from which the HTTP request body can be read for PUT and POST requests.

`wsgi.errors`: This is a text mode output stream (a file-like object) to which error output can be written. Applications should use `\n` as a line ending and assume it will be converted to the correct line ending. For many servers, `wsgi.errors` will be the server's main error log.

`wsgi.multithread`: This evaluates to `true` if the application object can be simultaneously invoked by another thread in the same process; it evaluates to `false` otherwise. It typically takes the value `0` or `1`.

`wsgi.multiprocess`: This evaluates to `true` if an equivalent application object can be simultaneously invoked by another process, and it evaluates to `false` otherwise. It typically takes the value `0` or `1`.

`wsgi.run_once`: This evaluates to `true` if the server or gateway expects that the application will be invoked only this one time during the life of its containing process. Normally, this will be `true` only if your Pylons application is being run from a CGI script.

Once again, you will rarely need to access WSGI variables directly because the components in Pylons that add them also present a clean API for their use, but it is useful to know they are there.

To see all the environment variables that Pylons sets up for you, add another action called `environ()` to the `HelloController` you created earlier so that it looks like this:

```
class HelloController(BaseController):

    def index(self):
        return 'Hello from the index() action!'
```

```
def environ(self):
    result = '<html><body><h1>Environ</h1>'
    for key, value in request.environ.items():
        result += '%s: %r <br />'%(key, value)
    result += '</body></html>'
    return result
```

If you visit `http://127.0.0.1:5000/hello/environ`, you will see all the keys and values displayed, including the WSGI variables and the traditional CGI-style environment variables.

Understanding the Pylons Request and Response

Now that you have learned about the fundamentals of HTTP and the environment, let's return to learning about Pylons, specifically, the `request` and `response` objects.

The `request` and `response` objects together represent all the information Pylons receives about the HTTP request and all the information Pylons is going to send in the response. Let's start by looking at the `request` object.

Request

The Pylons `request` object is actually a subclass of the `webob.Request` class provided by the WebOb package, which was automatically installed as one of Pylons' dependencies. A new instance of the class is created on each request based on the HTTP information Pylons receives via the environment. The object is available as the `pylons.request` object and is automatically imported at the top of any controllers you create with the `paster controller` command.

You can look at the API reference on the WebOb web site at `http://pythonpaste.org/webob/` for full details of the `Webob.Request` object's API and on the Pylons web site at `http://docs.pylonshq.com/modules/controllers_util.html#pylons.controllers.util.Request` for details of the subclass, but there are a few methods and attributes that are particularly worth mentioning now:

`request.environ`: You've already seen this when I discussed the environment. It is a dictionary that contains CGI-style environment variables, WSGI variables, and request header information, but it is not normally used directly. Instead, other attributes of the `request` global are used to obtain Python representations of the information it contains.

`request.headers`: This is a dictionary representing all the HTTP request headers. The dictionary is case insensitive.

`request.method`: This is the HTTP method used to request the URL; typically, it is GET or POST but could be PUT, DELETE, or others.

`request.GET`: This is a dictionary-like object with all the variables in the query string.

`request.POST`: This is a dictionary-like object with all the variables in the request body. This has variables only if the request was a POST and it is a form submission.

`request.params`: This is a dictionary-like object with a combination of everything in `request.GET` and `request.POST`. You will generally use `request.params` rather than `request.GET` or `request.POST`, although they all share the same API. I'll cover `request.params` more closely in a minute because it is the main object you will use to deal with form submissions.

`request.body`: This is a file-like object representing the body of a POST request.

`request.cookies`: This is a dictionary containing the cookies present.

`request.url`: This is the full request URL, with the query string, such as `http://localhost/app-root/doc?article_id=10`. There are also other attributes and methods for obtaining different parts of the URL and even for generating URLs relative to the current URL.

The request object also has attributes for most of the common HTTP request headers, such as request.accept_language, request.content_length, and request.user_agent. These properties expose the parsed form of each header for whatever parsing makes sense. For instance, request.if_modified_since returns a datetime object (or None if the header was not provided).

Although the request object has plenty of useful attributes and methods, the one you are likely to use the most is request.params. This contains a MultiDict object representing all the GET and POST parameters of the request. Of course, you can access the GET and POST parameters separately via the request.GET and request.POST attributes, but most of the time you are unlikely to be trying to obtain GET and POST parameters at once, so you can just use request.params that combines the data from each.

Note A MultiDict object is a dictionary-like object defined in the WebOb package that allows multiple values with the same key.

The request.params object can be treated in a number of ways. Imagine you've visited the URL http://localhost:5000/hello/index?a=1&a=2; you could then use the request.params object in the following ways:

```
>>> request.params
MultiDict([('a', '1'), ('a', '2')])
>>> request.params['a']
'1'
>>> request.params.get('a', 'Not present')
'1'
>>> request.params.get('b', 'Not present')
'Not present'
>>> request.params.getall('a')
['1','2']
```

Caution If you are used to programming with Python's cgi module, the way request.params works might surprise you because if the request has two parameters with the same name, as is the case with our example variable a, using request.params['a'] and request.get('a') returns only the first value rather than returning a list. Also, the methods described return the actual value, not an object whose .value attribute contains the value of the parameter.

There is also one other method, request.params.getone(), which returns just one value for the parameter it is getting. In this case, calling request.params.getone('a') raises an error because there is more than one value for a:

```
>>> request.params.getone('a')
Traceback (most recent call last):
...
    raise KeyError('Multiple values match %r: %r' % (key, v))
KeyError: "Multiple values match 'a': ['1', '2']"
```

To avoid problems obtaining just one value for a parameter when you expected many or obtaining many when you expected just one, you are encouraged to use the request.params.getone() and request.params.getall() methods in your own code rather than the dictionary-like interface.

Response

Now that you've looked in detail at the `request` object, you can turn your attention to the `response` object. You have actually been implicitly using the `response` object already if you've been following the examples in this chapter. Any time you return a string from a controller action, Pylons automatically writes it to the `response` object for you. Pylons then uses the `response` object to generate the HTTP information it returns to the browser, so any changes you make to the `response` object affect the HTTP response returned. Let's look at the `response` object in more detail.

The `response` object is also a subclass of `webob.Response`, and once again you can find full details on the WebOb web site and the Pylons documentation web site at the same URLs as mentioned for the `request` object, but there are some features worth drawing your attention to here.

The `response` object has three fundamental parts:

`response.status`: This is the response code plus message, like `'200 OK'`. To set the code without the reason, use `response.status_int = 200`.

`response.headerlist`: This is a list of all the headers, like `[('Content-Type', 'text/html')]`. There's a case-insensitive dictionary-like object in `response.headers` that also allows you to access these headers as well as add your own.

`response.app_iter`: This is an iterable (such as a list or generator) that will produce the content of the response. You rarely need to access this in a Pylons application, though, because Pylons automatically uses this to produce the content of the response for you.

Everything else derives from this underlying state. Here are the highlights:

`response.content_type`: This is the content type not including the `charset` parameter.

`response.set_cookie(key, value, max_age=None, path='/', domain=None, secure=None, httponly=False, version=None, comment=None)`: This sets a cookie. The keyword arguments control the various cookie parameters. The `max_age` argument is the length for the cookie to live in seconds (you can also use a `datetime.timedelta` object). The `Expires` key will also be set based on the value of `max_age`.

`response.delete_cookie(key, path='/', domain=None)`: This deletes a cookie from the client. This sets `max_age` to `0` and the cookie value to `''`.

Looking at the `HelloWorld` example from earlier in the chapter, you might have noticed that although the `Content-type` header sent to the browser was `text/html`, the message you returned was actually plain text, so the `Content-type` header should have been set to `text/plain`. Now that you have learned about the `response` object, you can correct this. Update the `index()` action of the hello controller so that it uses this response:

```
def index(self):
    response.content_type = 'text/plain'
    return 'Hello from the index() action!'
```

The server will reload when you save the change, and you can test the example again by visiting `http://localhost:5000/`. This time, the browser treats the message as plain text instead of HTML. If you are using the Firefox browser, you may notice it uses a different font to display the message this time.

Understanding Pylons Globals

The `request` and `response` objects you learned about in the previous section are referred to as Pylons *globals* because Pylons takes great care behind the scenes to make sure they can be used throughout your application including in your project's controllers and its templates. In this section, I'll cover all the other Pylons globals you can use in your application. Let's start by looking at the objects made available by default in your controllers.

If you look at the `controllers/hello.py` file in the `HelloWorld` project I've been using as an example, you will see the following imports at the top:

```
from pylons import request, response, session, tmpl_context as c
from pylons.controllers.util import abort, redirect_to
```

These lines are for importing the core Pylons globals, but in addition to these globals, there are also some other imports Pylons developers can add to their controllers to import optional Pylons globals:

```
import helloworld.lib.helpers as h
from helloworld.lib import app_globals
from pylons import config
```

You've already learned about the `request` and `response` globals, and I'll cover `h`, `app_globals`, and the template context object `c` in detail in the following sections, so let's just concentrate on `session`, `abort`, `redirect_to`, and `config` for the time being:

`abort(status_code=None, detail="", headers=None, comment=None)`: Occasionally you might want to immediately stop execution of the request and return a response bypassing the normal flow of execution of your application. You can do this with the `abort()` function, which is used like this:

```
def test_abort(self):
    username = request.environ.get('REMOTE_USER')
    if not username:
        abort(401)
    else:
        return"Hello %s"%username
```

In this example, if no `REMOTE_USER` environment variable is set, it means that no user has signed in, so the request is immediately aborted and returns a 401 status code (the correct HTTP status code for when a user has not been authenticated and is therefore not authorized to see a particular resource).

The function also allows you to add some text to form part of the status as well as any extra headers that should be set. If you use a 300 status code, the `detail` option should be the location to redirect to, but you would be better using the `redirect_to()` function in such circumstances.

Internally, the `abort()` function uses WebOb HTTPExceptions, which you'll learn about in Chapter 17.

`config`: This contains configuration information about the running application. It is described at `http://docs.pylonshq.com/modules/configuration.html`.

`redirect_to(*args, **kwargs)`: This function takes the same arguments as the `url_for()` function you will learn about in the next section. It allows you to specify a URL to redirect to by stating the routing variables that you want the URL to produce once it is called.

It is also possible to specify the HTTP status code you want to use with the `_code` argument.

For example:

```
redirect_to(controller='hello', action='other_action', _code=303)
```

`session`: This is a proxy to the Beaker session object that can be used as a session store with various back ends. By default, the session information is stored in your project's `data` directory, and cookies are used to keep track of the sessions. You'll learn how to use sessions in Chapter 8 when you implement a flash message system.

In Chapter 11, you will also learn about four more Pylons globals named `translator`, `ungettext()`, `_()`, and `N_()`, but these are too advanced for now.

Helpers

Helper functions (or *helpers* as they are known) are a concept borrowed from Ruby on Rails and are simply useful functions that you will find yourself using over and over again to perform common tasks such as generating form fields or creating links. Helper functions are all kept in the `lib/helpers.py` module in your project directory structure, but you must manually import them into your controllers if you want to use them. Since helpers are used frequently, it is common to import them using the shortened module name `h` rather than the full name `helpers` to save on typing:

```
import helloworld.lib.helpers as h
```

One of the most useful helpers is `h.url_for()`, which is a function that comes from the Routes package to help you generate URLs. You've already seen how the Routes system maps a URL to a particular controller action. Well, the `h.url_for()` helper does the reverse, mapping a controller and action to a URL. For example, to generate a URL for the `environ` action of the `hello` controller, you would use the following code:

```
h.url_for(controller='hello', action='environ')
```

In Pylons it is generally considered bad practice to ever write a URL manually in your code because at some point in the future you might want to change your URL structure, or your Pylons application might be mounted at a different URL. By using `h.url_for()` to generate your URLs, you avoid these problems since the correct URL is always generated automatically.

The Routes system is actually extremely powerful, and you will see some of the details of the way it works in Chapter 9. One aspect worth mentioning now is that you can also specify extra variables in your route maps. For example, if you wanted URLs in the format `/calendar/view/2007/06/15`, you might set up a map that treats `calendar` as the controller and `view` as the action and then assigns `2007` to a variable `year`, `06` to a variable `month`, and `15` to a variable called `day`. When Pylons called your controller action, it would pass in `year`, `month`, and `day` as the parameters, so your action would look like this:

```
def view(self, year, month, day):
    return "This is the page for %s/%s/%s"%(year, month, day)
```

In this way, the URL itself can contain important information about what the page should display, and this both saves you having to pass such variables around your application as hidden fields and means your URL structure much better matches what is actually going on in your application.

You can use many other useful helpers to make your programming easier, but one of the great things about Pylons is that you can easily add your own too. Adding a new helper to the `h` object is as simple as importing a new function into your project's `lib/helpers.py` module or defining a new object in it. As an example, let's refactor the code you wrote earlier to print the environment into a useful helper. At the end of your project's `lib/helpers.py` file, add the following function:

```
def format_environ(environ):
    result = []
    keys = environ.keys()
    keys.sort()
    for key in keys:
        result.append("%s: %r"%(key, environ[key]))
    return '\n'.join(result)
```

Then after importing the helpers as `h`, you can update your action to look like this:

```
def environ(self):
    response.content_type = 'text/plain'
    return h.format_environ(request.environ)
```

■ **Tip** In this case, the helper function needed access to the environment that formed part of the request, so if you are a keen object-oriented programmer and had wanted to refactor the environment formatting code, you might have been tempted to add a private method to the controller rather than create a helper function that needs the request information passed in. Generally speaking, it is a lot better to add useful code that you intend to use a lot as a simple helper object than to add methods to controller classes because then they can easily be accessed throughout your Pylons application rather than just in controller actions.

You'll see more of the built-in Pylons helpers when I cover form handling in Chapter 6.

Context Object

When you develop a real Pylons application, you will quickly find you need to pass request-specific information to different parts of your code. For example, you might need to set some variables to be used in a template or in a form validator.

Because Pylons is designed to work in a multithreaded environment, it is important that this information is passed around your application in a thread-safe way so that variables associated with one request don't get confused with variables from any other requests that are being executed at the same time.

Pylons provides the context object `tmpl_context` for precisely this purpose, and again, since it is used so frequently, it is imported by default into your controllers as the `c` object. The `c` object is a `StackedObjectProxy` that always returns the correct data for the current request.

You can assign attributes to the `c` object like this, and they will be available throughout the application:

```
c.my_data = 'Important data'
```

You can choose any attribute name you want to assign variables to as long as they don't start with the _ character and are a valid Python name. They will then be available throughout your templates and application code and will always contain the correct data for the thread that is handling a particular request.

As was mentioned in the discussion about helpers, the Routes system Pylons uses allows you to use parts of the URL as variables in your application. Since you frequently need to access these variables in templates and other areas of your code, Pylons automatically sets up the `c` object to have any of the Routes variables that your action specified attached to it. You could therefore modify the helpers example from earlier to look like this, and it would still work in the same way:

```
def view(self, year, month, day):
    return "This is the page for %s/%s/%s"%(c.year, c.month, c.day)
```

There is one more important aspect to learn about the `c` variable that you might not expect at first. If you access an attribute that doesn't exist, the value returned will be the empty string `''` rather than an `AttributeError` being raised as you might expect. This enables you to write code such as this without needing to test each part of the statement to check that the attribute exists:

```
data = c.some_value or c.some_other_value or "Not specified"
```

This code will set `data` to be `c.some_value` if it exists and otherwise `c.some_other_value` if that exists; if neither exist, the value will be set to `"Not specified"`.

This style of code can be useful occasionally, but if you are not used to writing code like this, I strongly recommend you stick to explicitly testing values of attributes you are not sure about to avoid the risk of errors. You can do this with the Python functions `hasattr()` and `getattr()`, which are used like this:

```
if hasattr(c, "foo"):
    x = c.foo
else:
    x = 'default'
y = getattr(c, "bar", "default")
```

To use the strict version of the `c` global, edit your project's `config/environment.py` file, and add the following line just before the lines to customize your templating options:

```
config['pylons.strict_c'] = True
# Customize templating options via this variable
```

With the `strict_c` option enabled, `c` will raise an `AttributeError` as you would expect. It is strongly recommended you set the `strict_c` option if you are a Pylons beginner.

■ **Caution** If you come across an error you can't quite understand when performing some operation on an attribute of `c`, it is possible that you have forgotten to specify the attribute and that the empty string is being returned instead. Trying to perform an operation on the string when you expected it to be a different object may be what is causing the error.

App Globals Object

Sometimes you might want information to be available to all controllers and not be reset on each request. For example, you might want to set up a database connection pool when the application is loaded rather than creating a connection on each request. You can achieve this with the `app_globals` object.

■ **Note** In previous versions of Pylons, the `app_globals` global was simply named `g`. New Pylons applications should use the full name `app_globals` instead.

The `app_globals` variable is actually just an instance of your `Globals` class in your application's `lib/app_globals.py` file. It gets set up in the `config/environment.py` file of your project.

Any attributes set to `self` in the `__init__()` method of the `Globals` class will be available as attributes of `app_globals` throughout your Pylons application. Any attributes set on `app_globals` during one request will remain changed for all subsequent requests, so you have to be very careful not to accidentally change any global data by mistake.

Here is a simple counter example that demonstrates how the `app_globals` object works. First modify your `lib/app_globals.py` `Globals` class so that the `__init__.py` method looks like this:

```
def __init__(self):
    self.visits = 0
```

You will now be able to access the `visits` attribute as `app_globals.visits` in your controllers. First import the global from Pylons:

```
from pylons import app_globals
```

Next add a new action to the end of the `HelloController`:

```
def app_globals_test(self):
    app_globals.visits += 1
    return "You are visitor number %s." % app_globals.visits
```

If you restart the Paste HTTP server and visit `http://localhost:5000/hello/app_globals_test`, you should see the message `You are visitor number 1`. If you visit the page again, the message will be changed to `You are visitor number 2`. On each subsequent request, the counter will increase.

If you restart the server, you will see that the value of `app_globals.visits` is reset to 0 because a new instance of the `helloworld.lib.app_globals.Globals` class is created when the server starts.

■ **Caution** Because the `app_globals` object is persistent across requests, it can be modified by any thread. This means it is possible that a value attached to `app_globals` might be different at the start of a request than at the end if another thread has changed its value in between. This doesn't matter in our example, but it is something to be aware of if you aren't used to working in multithreaded environments.

Configuring Pylons

You configure Pylons applications in two places. The first is the configuration file you saw earlier in the chapter when you looked at the `HelloWorld` project's `development.ini` file. The configuration file is where any per-instance configuration options should be put. For example, the port and hostname might vary between a production deployment of your Pylons application and a development installation, so these are options that are set in the configuration file.

Any configuration that should affect the whole application and should not be customizable on a per-instance basis is set in Python code in one of the files in your project's `config` directory. These are the following:

`middleware.py`: This is where the Pylons application is constructed and where you can change the Pylons middleware stack itself. Middleware is considered an advanced topic and is dealt with in Part 3 of the book.

`routing.py`: This is where you define your application's routes. You saw a simple example earlier in the chapter and will look at routing in much more detail in Chapter 9.

`environment.py`: This is where you configure most of Pylons' internal options, although you generally don't need to because the defaults are very sensible. One time when you might need to set some options in this file is if you want to use a different templating language to the Pylons default, which is Mako. You'll learn about this in Chapter 5.

Once you have configured your application, whether in the configuration file or the `config/environment.py` file, you will want to be able to access that configuration in your Pylons application. All configuration is contained in the `config` global, which can be imported into a controller like this:

`from pylons import config`. The `config` object has a number of attributes representing different aspects of Pylons configuration. These are described in detail at `http://docs.pylonshq.com/modules/configuration.html`, but these are two that you are likely to use frequently:

`config.app_conf`: This is a dictionary of all the application options set in the `[app:main]` part of the config file being used including any custom options used by your application. For example, to access the location of the cache directory, you could use `config.app_conf['cache_dir']`.

`config.global_conf`: This is very similar to `config.app_conf`, but rather than providing a dictionary interface to the `[app:main]` section of the config file, this attribute represents the global options specified in the `[DEFAULT]` section.

There are also attributes for the package name, default character set, paths to look for templates, and more. See the documentation for full details.

Controller Imports

In addition to the Pylons globals I've described, there are a number of other useful objects that Pylons provides for you to use in your controllers. Each of these objects is described in detail in the Pylons module documentation, which you can browse online at `http://docs.pylonshq.com/modules/index.html`, but they are worth mentioning here so that you are aware of them.

First let's look at the available decorators:

`pylons.decorators.jsonify(func)`: Given a function that will return content, this decorator will turn the result into JSON, with a content type of `text/javascript`, and output it.

`pylons.decorators.validate(...)`: This validates input either for a FormEncode schema or for individual validators. This is discussed in detail in Chapter 6.

`pylons.decorators.secure.authenticate_form(func)`: This decorator uses an authorization token stored in the client's session for prevention of certain cross-site request forgery (CSRF) attacks.

`pylons.decorators.secure.https(*redirect_args, **redirect_kwargs)`: This decorator redirects to the SSL version of a page if not currently using HTTPS.

`pylons.decorators.rest.dispatch_on(**method_map)`: This dispatches to alternate controller methods based on the HTTP method.

`pylons.decorators.rest.restrict(*methods)`: This restricts access to the function depending on the HTTP method. You'll see it used in Chapter 8.

`pylons.decorators.cache.beaker_cache(...)`: This cache decorator utilizes Beaker. This caches the action or other function that returns a "pickleable" object as a result.

Pylons also provides two different types of controllers:

`pylons.controllers.core.WSGIController`: This is the controller that your project's `BaseController` is inherited from and is the only controller you will use in this book.

`pylons.controllers.xmlrpc.XMLRPCController`: This controller handles XML-RPC responses and complies with the XML-RPC specification as well as the XML-RPC Introspection specification. It is useful if you want to build XML-RPC web services in Pylons.

Finally, it is worth mentioning three more objects you are likely to use a lot in your controllers, but each of these has their own chapter and will be dealt with later in the book:

`render`: This is used for rendering templates and is discussed in Chapter 5.

`model`: This is used for the model component of your application and is discussed in Chapter 7.

`log`: This is used to output log messages.

In its simplest form, you can simply write `log.error('Log this message')` in a controller, and the message will be logged to the console of the Paste HTTP server. Logging is described in detail in Chapter 20.

Summary

This chapter covered quite a lot of ground, including using static files, understanding HTTP and the environment, using the `request` and `response` objects, and understanding the basics of Routes. In the next chapter, you'll take a detailed look at how to track down problems when they occur and how to handle them so that the user is presented with an appropriate error message. After that, you'll be ready to get properly stuck into Pylons development, so you'll learn how to use templates before looking at how forms are handled.