CHAPTER 9

URLs, Routing, and Dispatch

All web applications and frameworks need some mechanism for mapping the URL that a user enters in the browser to the code that should be executed when the server receives the request. Different languages and frameworks take different approaches to this. In PHP- or CGI-based systems, the URL represents a path on the hard disk to the file that should be executed. In Zope, URLs are treated as paths in an object hierarchy, and in Django, URLs are matched to code based on regular expressions.

Pylons doesn't use any of these approaches. Instead, it uses a very powerful and flexible system called Routes, which is an improved version of the Ruby on Rails routing system. Routes allows you to quickly and easily specify how groups of similar URLs map to your controllers; in turn, this allows you to create whatever URL structure you prefer for your application. Unlike other lookup systems, the URL is completely decoupled from the code that handles it. This also makes it easier to change your URL structure if you need to do so.

You've already seen Routes in action in previous chapters. For example, in Chapter 3 you saw a route that looked like this:

```
map.connect('/{controller}/{action}/{id}')
```

Each of the labels between curly brackets represents dynamic parts of a route. When a route is matched, the corresponding URL parts are matched, and their values are assigned to routing variables corresponding to the labels of the dynamic parts within the curly brackets. In this way, the one route can be used to match a large range of URLs.

You can also use Routes to generate URLs with `h.url_for()`. To do this, you need to import the Routes `url_for()` function into your project's `lib/helpers.py` file:

```
from routes import url_for
```

Here's an example of how to use `h.url_for()`, which results in the URL `/page/view/1` being generated:

```
h.url_for(controller='page', action='view', id=1)
```

As you can see, the arguments to `h.url_for()` correspond to the routing variable names used when defining the route.

When a user visits this URL, the same route used for generating the URL will also be used for matching it. In this case, the URL generated by the earlier call to `h.url_for()` will also generate the routing variables `controller='page'`, `action='view'`, and `id=1` when the URL is visited, and this results in the page controller's `view()` action being called with the ID `1`.

You can also give individual routes a name by specifying a string as the first argument to `map.connect()`. This reduces the amount of typing you have to do when generating URLs. For example, you might define this route:

```
map.connect('blog_entry', '/blog/view/{id}')
```

You could then generate a URL to the entry with ID 1 like this:

```
h.url_for('blog_entry', id=1)
```

Now that you have a rough feel for how Routes works, let's dive in straightaway and look at how Routes is set up and used in Pylons and how Pylons uses information from Routes to dispatch requests.

■ **Caution** One problem with Routes is that because it was originally ported from the Ruby on Rails implementation, it still contains a number of features that you might have used when developing a Pylons 0.9.6 project that are no longer considered good practice. In the section "Unnecessary Routes Features" later in this chapter, I'll explain why these features are no longer recommended. It is likely they will be removed in a future version of Routes, so it is important you avoid using them.

Pylons Routing in Detail

At its heart Routes is all about two things: analyzing a URL to produce a list of variables and being able to re-generate that URL from the same variables. These variables are known as *routing variables*, and they are used by Pylons to determine which controller action should be called and the arguments it should be called with. The URL matching is done by comparing the URL to a series of strings known as *route paths*, which, together with a set of options, define how a particular set of URLs can be turned into routing variables and back to URLs again. The route paths and options together are known as a *route*, and a set of routes is known as a *route map*.

Let's see how these concepts are applied in a Pylons application. You can find your project's route map in its `config/routing.py` file. Let's use the SimpleSite route map as an example. It looks like this:

```
def make_map():
    """Create, configure and return the routes Mapper"""
    map = Mapper(directory=config['pylons.paths']['controllers'],
                 always_scan=config['debug'])
    map.minimization = False

    # The ErrorController route (handles 404/500 error pages); it should
    # likely stay at the top, ensuring it can always be resolved
    map.connect('/error/{action}', controller='error')
    map.connect('/error/{action}/{id}', controller='error')

    # CUSTOM ROUTES HERE

    map.connect('/{controller}/{action}')
    map.connect('/{controller}/{action}/{id}')

    return map
```

Here you can see that the `make_map()` function is responsible for creating a route map object called `map` and returning it. The routes are added to the map via the `map.connect()` function, which takes a route path such as `'/{controller}/{action}/{id}'` and some optional arguments called the *route options*. Each of the labels for the dynamic parts between the { and } characters specify the positions of routing variables that are matched against any value in the corresponding position

within a URL. When a URL is entered, Pylons tries to match it by calling `map.match(url)`. Routes then searches each of the routes in the route map from top to bottom until it finds a route that matches the URL. Because matching is done from top to bottom, you are always advised to put your custom routes below the ones that Pylons provides to ensure you don't accidentally interfere with the default behavior of Pylons. More generally speaking, you should always put your most general routes at the bottom of the route map so that they don't accidentally get matched before a more specific route lower down in the route map.

To demonstrate the behavior of the route map as it stands, let's consider some example URLs and the routing variables that are produced when the URL is matched. Let's start with the site root URL, which is `/`:

```
URL:    /
Result: None
```

As you can see, there are no routes with just one `/` character in them, so the URL `/` is not currently matched. If you visited this URL, you would see the `index.html` file served from your project's `public` directory or a 404 Not Found error page. You'll add a route to handle the root URL in the second part of the SimpleSite tutorial in Chapter 14.

```
URL:    /page/view/1
Result: {'action': 'view', 'controller': 'page', 'id': '1'}
```

As you already know from having tested the SimpleSite application in Chapter 8, this URL is matched by the last route in the route map and results in a page being displayed. The URL fragment `page` is matched to `{controller}`, `view` is matched to `{action}`, and `1` is matched to `{id}`.

Now consider the URL `/error/img`:

```
URL:    /error/img/logo.png
Result: {'action': 'img', 'controller': 'error', 'id': 'logo.png'}
```

This is matched by the second route, `map.connect('/error/{action}', controller='error')`, because the first part of the URL begins with `/error/`. The `img` part is matched to `{action}`, and `logo.png` is matched to `{id}`. Notice that the route path doesn't contain a `{controller}` part, though, so you might wonder how the `controller` routing variable gets assigned a value. You'll see in the route map that the error route definition also takes a default argument of `controller='error'`. This means that if the route is matched, the `controller` routing variable will be automatically added to the results with a value of `'error'`. You'll learn about how Pylons uses this route in its internal handling of error documents in Chapter 19.

So far, this should all seem fairly intuitive; URLs are matched against static text or dynamic parts representing routing variables in the route path, and default values can be added when needed. Let's try another example:

```
URL:    /section/view/1
Result: None
```

This time the URL isn't matched even though at first glance it looks like it should be. To understand why, you need to know a little bit more about how Routes works.

The `Mapper` object takes a number of arguments, including the following:

`controller_scan`: This takes a function that will be used to return a list of valid controllers during URL matching. If the `directory` argument is also specified, it will be passed into the function when it is called.

`directory`: This is passed into `controller_scan` for the directory to scan. It should be an absolute path if using the default `controller_scan` function.

`always_scan`: This specifies whether the `controller_scan` function should be run during every URL match. This is typically a good idea during development so the server won't need to be restarted when a controller is added, but you might want to disable it on production systems to avoid the small overhead of the check. After all, you are unlikely to have specified routes to controllers that don't exist in production code.

`explicit`: This has a default value of `False`, but when it is set to `True`, some of the features of Routes such as route memory and implicit defaults are disabled. You'll learn about these features and why you should usually set this option to `True` in the "Unnecessary Routes Features" section.

Pylons uses the Routes default `controller_scan()` function, which will scan the directory specified by the `directory` argument to check for controllers. If a particular route matches a URL but the controller resulting from the match doesn't exist, the route will be ignored, and the next route will be tried. In this case, the `directory` argument is set to `config['pylons.paths']['controllers']`, which contains the path to your project's `controllers` directory. Since no `section` controller exists yet, the route in the previous example didn't match. For production use, you don't need to have Routes scan the controller directory on each match because the controllers aren't likely to change, so the `always_scan` variable is automatically set to the same value as the value of the `debug` option in the config file, enabling the scan in development mode and disabling it in production mode.

■ **Note** You are very unlikely to want to customize the `controller_scan()` function in your own application, but I've mentioned it here so that you are aware of its behavior when you see it crop up in some of the examples in this chapter.

By carefully defining routes and specifying their order in the *route map*, it is possible to quickly and easily map quite complex URL structures into simple sets of routing variables that can be used in your application. Before you go on to look at the details of how to write your own routes, let's take a brief look at how Pylons uses the routing variables to call a particular controller action.

Pylons Dispatch

Once a URL has been matched against a route and the controller has been verified to exist, Pylons checks to see whether it can find the action.

Pylons is set up so that any controller methods that start with an _ character are not considered actions at all and cannot be called as a result of a URL dispatch. If the action specified in the routing variables doesn't exist or it starts with an _ character, then the check fails. In debug mode for development, this raises an exception to explain the mistake; in production mode, it simply results in a 404 Not Found error page being displayed.

If the controller and action both exist, then Pylons looks for a `__before__()` method attached to the controller class. If the method exists, it is executed. The `__before__()` method can therefore be used to set up per-request objects or perform other processing to be carried out before each action in the controller is called.

Next, Pylons inspects the action to see what arguments it takes and calls it in such a way that any arguments that have the same names as the routing variables are called with the value of the routing variable as the argument. Here are some examples:

```
# No routing variables used in the action
def no_routing_variables(self):
    return 'No routing variables used'
```

```
# Populate the ``id`` with the value of the ``id`` routing variable
def id_present(self, id):
    return 'The id is %s' % id
```

Tip You've already seen routing variables being used in this way in the `page` controller from the SimpleSite tutorial in Chapter 8, and you'll recall from Chapter 3 that you can also retrieve the `controller` name and `action` in the same way. All the routing variables that are passed to the action are also automatically added as attributes to the template context global `c`.

In addition to routing variables, you can also specify both `environ` and `start_response` as arguments in your actions. The `environ` argument will get populated with the same WSGI environment dictionary you can access via the Pylons request global as `request.environ`, and the `start_response()` callable is a WSGI callable you'll learn about in Chapter 16. Both `environ` and `start_response` get added as attributes to the template context global `c` in the same way as the routing variables do if you use them in an action. To see all these features in action, consider the following example. When the `test()` action is called, the browser will display `True, True, True, True`:

```
def __before__(self, id):
    c.id_set_in_before = id

def test(self, environ, start_response, controller, id):
    w = c.id_set_in_before == id
    x = c.environ == environ == request.environ
    y = c.start_response == start_response
    z = c.controller == controller
    request.content_type = 'text/plain'
    return "%s %s, %s, %s"%(w, x, y, z)
```

After an action is called, Pylons looks for an `__after__()` method attached to the controller, and if it exists, it is called too. You can use `__after__()` for performing any cleanup of objects you created in the `__before__()` method.

Both the `__before__()` and `__after__()` methods can also accept any of the routing variable names as arguments in exactly the same way as the controller action can.

Of course, there are occasions where it is useful to be able to access the routing variables outside your controller code. All the routing variables are available in the `request.urlvars` dictionary and are also available in `request.environ["wsgiorg.routing_args"]`.

Routes in Detail

Now that you have seen some examples of routes being matched against URLs and understand how Pylons uses the routing variables to dispatch to your controller actions, it is time to learn the details of how you can construct individual routes.

Route Parts

As you've already seen, a route is specified by arguments to `map.connect()` that include a route path and a number of options. The route path itself can consist of a number of different types of parts between `/` characters:

Static parts: These are plain-text parts of the URL that don't result in any route variables being defined. In the default Pylons setup, the string `error` in the route path `/error/{action}` is a static part.

Dynamic parts: The text in the URL matching the dynamic part is assigned to a routing variable of that name. Dynamic parts are marked in the route path as the routing variable name within curly brackets. In the default Pylons setup, the string `{action}` in the route path `/error/{action}` is a dynamic part. Dynamic parts can also be specified by the routing variable name preceded by a colon, which is the style used in earlier versions of Routes. For example, you could change the first error route path from `/error/{action}` to `/error/:action`, and it would still work.

Wildcard parts: A wildcard part will match *everything* (including / characters) except the other parts around it. Wildcard parts are marked in the route path by preceding them with the * character. The default Pylons setup doesn't include any routes that use wildcard parts, but you can use them in your own routes if you need to do so.

To demonstrate the different types of route parts, consider the following URL:

```
/wiki/page/view/some/variable/depth/file.html
```

and a route path to match it, which includes a wildcard part:

```
/wiki/{controller}/{action}/*url
```

In this case, `wiki` is treated as a static part, and `page` is a dynamic part that is assigned to the `controller` route variable. In Routes, / characters separating dynamic or wildcard parts are not treated as static parts, so the next / is ignored. `view` is a dynamic part assigned to the `action` route variable. The following / is also ignored, and the whole of the rest of the URL is considered a wildcard part and assigned to the `url` route variable.

Any URL starting with `/wiki/` followed by two sets of characters, each followed by a / character, will be matched by this route path as long as the controller exists.

Let's test this with a simple example. Rather than modifying the SimpleSite project, you can set up your own route map directly. You just have to specify a dummy `controller_scan` function to return a list of valid controllers. In the example, `map.minimization` is set to `False` so that the route map is set up in the same way it would be in a Pylons application. The `controller_scan()` function will just return `['error', 'page']` to simulate that only the `page` and `error` controllers exist. Here's what the route map looks like:

```
>>> def controller_scan(directory=None):
>>>     return ['error', 'page']
...
>>> from routes import Mapper
>>> map = Mapper(controller_scan=controller_scan)
>>> map.minimization = False
>>> map.connect('/wiki/{controller}/{action}/*url')
>>> print map.match('/wiki/page/view/some/variable/depth/file.html')
{'action': u'view', 'url': u'some/variable/depth/file.html', 'controller': u'page'}
```

The URL is matched as expected. Let's try to match some different URLs:

```
>>> print map.match('/some/other/url')
None
>>> print map.match('/wiki/folder/view/some/variable/depth/file.html')
None
```

Notice that if the URL doesn't match or if the matched controller doesn't exist, the route is not matched, and the `map.match()` function returns `None`.

The Pylons implementation of Routes doesn't require the dynamic and wildcard portions of the route path to be separated by a `/` character. Here you're using a `.` character as a separator. Note that you need to create a new `Mapper` to set this new route because you have already called `map.connect()` on the previous one. Once again, you use the dummy `controller_scan()` function and set `map.minimization = False` to turn off minimization in the same way as the Pylons setup:

```
>>> map = Mapper(controller_scan=controller_scan)
>>> map.minimization = False
>>> map.connect('/blog/{controller}.{action}.*url')
>>> print map.match('/blog/page.view.some/variable/depth/file.html')
{'action': u'view', 'url': u'some/variable/depth/file.html', 'controller': u'page'}
```

As these examples have demonstrated, you can think of wildcard parts as being very much like dynamic parts but also matching `/` characters.

Although wildcard parts are a powerful tool, they generally aren't used very often and can be a sign that you haven't designed your routes very well. The risk is that because they match any character including `/`, they will match very many URLs. If you were to have a wildcard route near the top of your route map, it could easily match URLs for which there was a more specific route later in the route map. The vast majority of the time you will just use static and dynamic parts in your route paths, but if you do use wildcard routes, they should always be placed near the bottom of the route map to avoid them accidentally matching URLs when there is a more specific route later in the map.

These are three cases when you might choose to use wildcard routes:

- When an entire portion of the URL is going to be passed to another application (for example, Paste's static file server)
- When you are trying to match a string ID that might include `/` characters
- When you want to pass the entire request URL to a particular action to handle an error condition, rather than allowing a 404 Not Found error page to be returned

Default Variables

With the knowledge you've already gained, you will be able to create most of the routes you are likely to need, but there is one problem. Every route matched must result in at least the routing variables `controller` and `action` being assigned a value, but URLs such as `/` don't contain enough characters to be able to deduce the controller and action names from them.

To solve this problem, Routes allows you to specify default values for any routing variables as part of the routing map's `connect()` function. You've already seen default variables in action during the discussion of Pylons' error route, but let's look at another example:

```
>>> map = Mapper(controller_scan=controller_scan)
>>> map.minimization = False
>>> map.connect('/archives/by_eon/{century}', controller='page', action='list')
```

In this example, the `controller` and `action` both have default values, so they are called *default variables*. You'll notice that you can't create a URL that will affect the value of the `controller` and `action` variables since they aren't present in the route path, so their default values of `'archives'` and `'list'` will always be used. Such variables are known as *hard-coded variables*.

The value assigned to the `century` routing variable will depend on the URL with which it is matched:

```
>>> print map.match('/archives/by_eon/')
None
>>> print map.match('/archives/by_eon')
None
>>> print map.match('/archives/by_eon/1800')
{'action': u'aggregate', 'controller': u'page', 'century': u'1800'}
```

■ **Caution** One thing you have to be aware of when choosing names for routing arguments is that the `map.connect()` method also accepts certain keyword arguments including `requirement`, `_explicit`, `_encoding`, and `_filter`. If you choose a routing variable with the same name as one of these arguments, then you will not be able to specify a default value because Routes will not interpret the argument as a default value. To prevent this potential problem, future versions of Routes will not allow any routing variable names starting with an `_` character, so you would be wise to avoid starting your routing variables with an `_`.

You should avoid using two other names as routing variables; these are the WSGI objects `environ` and `start_response`. Both can be passed automatically to controller actions when they are called. If you choose routing variables with the same names as these, you won't be able to access the routing variables as arguments to the controller actions because the Pylons dispatch mechanism will assume you want the WSGI objects themselves rather than the routing variables of the same name.

It is possible to take the use of default variables to an extreme. A route where the controller and action are hard-coded is known as an *explicit route*. Here's an example:

```
map.connect('/blog/view/1', controller='blog', action='view', id=1)
```

Here the whole route path is made only from static parts, and all the routing variables are hard-coded as default variables. The benefit of this approach is that only one URL will ever match this route, but the drawback is that defining all your URLs this way requires adding an awful lot of routes to your application.

Generating URLs

Now that you have seen how Pylons uses Routes and you understand the details of how URLs are matched, you can turn your attention once again to how to generate URLs.

Routes includes two functions for use in your web application that you will find yourself using frequently:

- `h.url_for()`
- `redirect_to()`

You've already seen the `h.url_for()` function a few times, and you'll recall that to make it available in your controllers and templates, you need to import it into your project's `lib/helpers.py` file from the `routes` module.

The `redirect_to()` function is automatically imported into your controllers from the `pylons.controllers.util` module. It is used to redirect a controller to a different URL within your application. If you look at the page controller in the SimpleSite project you've been developing, you'll see that up to now redirects have been coded manually by specifying the correct HTTP status code for a redirect and setting a `Location` HTTP header using the `response` global. Here's the `save()` action as an example:

```
@restrict('POST')
@validate(schema=NewPageForm(), form='edit')
def save(self, id=None):
    page_q = meta.Session.query(model.Page)
    page = page_q.filter_by(id=id).first()
    if page is None:
        abort(404)
    for k,v in self.form_result.items():
        if getattr(page, k) != v:
            setattr(page, k, v)
    meta.Session.commit()
    session['flash'] = 'Page successfully updated.'
    session.save()
    # Issue an HTTP redirect
    response.status_int = 302
    response.headers['location'] = h.url_for(controller='page', action='view', I
       id=page.id)
    return "Moved temporarily"
```

You could instead replace the last five lines to use `redirect_to()` like this:

```
# Issue an HTTP redirect
return redirect_to(controller='page', action='view', id=page.id)
```

As you can see, this is much neater, so it is a good idea to use `redirect_to()` in your controllers rather than writing out the long version of the code. `redirect_to()` is commonly used after a form submission to redirect to another URL so that if the user clicks Refresh, the form data isn't sent again.

One of the major benefits of using `h.url_for()` and `redirect_to()` rather than generating your own URLs manually is that Pylons can automatically take account of the URL at which your application is deployed and adjust the URLs automatically.

As an example, imagine you have created a new Pylons project to manage application forms. You might choose to deploy the Pylons application at a URL such as `/forms` so that it appears to be part of an existing site. This means that when a user visits `/forms`, the Pylons application needs to know that they are really requesting a URL, which would be `/` if the Pylons application was running on a stand-alone server. This also means that any time the application creates a URL, it has to start `/forms`; otherwise, the links wouldn't point to the Pylons application. In this setup, the `SCRIPT_NAME` environment variable would be set to `/forms` so Pylons can work out that the URLs needed to be adjusted. Any time you call any of the Pylons URL generation functions, the URL generated is modified to take account of the `SCRIPT_NAME` environment variable, so all the URLs get generated correctly without needing any manual modification.

Named Routes

If you are regularly using the same route in your application, it can quickly become tedious to type all the routing variables every time you want to use `h.url_for()`. This is where *named routes* come in. A named route is like an ordinary route, but you access it directly via a name.

Here's an example:

```
map.connect('category_home', 'category/{section}', controller='blog', action='view',
          section='home')
```

Then in your Pylons application you might use this:

```
h.url_for('category_home')
```

This is equivalent to generating the URL using the routing variables in full, both generate the URL `/category/home`:

```
h.url_for(controller='blog', action='view', section='home')
```

As you can see, this saves a bit of typing. You can also specify keyword arguments, and it will override defaults associated with the route name. The following:

```
h.url_for('category_home', section='admin')
```

generates the URL `/category/admin` and is equivalent to this:

```
h.url_for(controller='blog', action='view', section='admin')
```

Static Named Routes

Routes also supports a feature called *static named routes*. These are routes that do not involve actual URL generation but instead allow you to quickly alias common URLs that are external to your site. To make a route static, you add `_static=True` as an argument. For example, if your route map contained this:

```
map.connect('google_search', 'http://www.google.com/search', _static=True)
```

you could now reference `http://www.google.com/search` by typing this in your application:

```
h.url_for('google_search')
```

This is especially useful if you are accessing resources throughout your application that you think might change because it means you have to change the URL in only one place. Static named routes are ignored entirely when matching a URL.

Internal Static Routes

You can also use `h.url_for()` to reference static resources in your project's `public` directory. Routes will automatically prepend the URL you give it with the proper `SCRIPT_NAME` so that even if you are running your Pylons application under a different URL, the correct URL will be generated. For example, if your Pylons application was mounted at `/forms` and you used this in your application, the URL `/forms/css/source.css` would be correctly generated with this code:

```
h.url_for('/css/source.css')
```

For portable web applications, it's highly encouraged that you use `h.url_for()` for all your URLs, even those that are static resources and images. This will ensure that the URLs are properly handled in any of the possible deployment setups you could use for your application.

Choosing Good URLs

Now that you've seen some of the ways you can use Routes to match and generate URLs, let's take a look at what actually makes a good URL structure.

First here's a definition of the parts of a URL:

```
http://jimmyg.org:80/some/url#fragment?foo=bar
|--|   |---------|--|--------|--------|------|
 |          |     |     |        |       |
protocol    |    port   |     fragment   |
       domain name  path info      query string
```

Now here are some tips:

Describe the content: An obvious URL is a great URL. If a user can glance at a link in an email and know what it contains, you have done your job. This means choosing URL parts that accurately describe what is contained at that level of the URL structure. It's usually better to use a descriptive word rather than an ID in the URL wherever possible. For example, if you were designing a blog, you should try to use `apr` instead of `04` to represent April, and you should use the name of a category rather than its ID if that is appropriate. Choosing URLs that describe their content makes your application intuitive to your users and gives search engines a better chance of accurately ranking the page.

■ **Tip** Edward Cutrell and Zhiwei Guan from Microsoft Research conducted an eye-tracking study of search engine use that found people spend 24 percent of their "gaze time" looking at the URLs in the search results. If your URLs accurately describe their content, users can make a better guess about whether your content is relevant to them.

Keep it short: Try to keep your URLs as short as possible without breaking any of the other tips here. Short URLs are easier to type or to copy and paste into documents and emails. If possible, keeping URLs to less than 80 characters is ideal so that users can paste URLs into email without having to use URL-shortening tools such as `tinyurl.com`.

Hyphens separate best: It is best to use single words in each part of a URL, but if you have to use multiple words, such as for the title of a blog post, then hyphens are the best characters to use to separate the words, as in `/2008/nov/my-blog-post-title/`. The `-` character cannot be used in Python keywords, so if you intend to use the URL fragments as Python controller names or actions, you might want to convert them to `_` characters first. Incidentally, using hyphens to separate words is also the most readable way of separating terms in CSS styles.

Static-looking URLs are best: Regardless of how your content is actually generated, it is worth structuring URLs so that they don't contain lots of `&`, `=`, and `?` characters that most visitors won't properly understand. If you can write a URL like `?type=food&category=apple` as `/food/apple`, then users can see much more quickly what it is about. Routes makes this sort of transformation easy, so there is no need to make extensive use of variables in query strings when they could form part of the URL.

Keeping URLs lowercase makes your life easier: The protocol and domain name parts of a URL can technically be entered in any case, but the optional fragment part after the # character is case sensitive. How a particular server treats anything between the two depends on the server, operating system, and what the URL resolves to. Since Pylons applications can be deployed on many different servers, you have to be aware of the potential problems. Unix filesystems are usually case sensitive, while Windows ones aren't. Mac OS X systems can be case sensitive or case insensitive depending on the filesystem. This means that if the URL resolves to a file, Windows servers will generally allow any case, while Unix ones won't. Query string parameters are also case sensitive. You can generally save yourself a headache by keeping everything lowercase and issuing a 404 for anything that isn't. Of course, if you are writing a wiki application where the page names depend on the capitalization, then you'll need to make the URLs case sensitive.

Keep the underlying technology out of the URL: Your users aren't likely to care which specific technology you are using to generate your pages or whether they are generated as HTML or XHTML, so the basic rule is don't use a file extension for dynamically generated pages unless you are doing something clever in your application internally like determining the format to represent the content based on the extension. It is generally best to choose names that represent what the URL *is* rather than its technology.

Never change a URL: Otherwise, your users won't be able to find the page they bookmarked, and any page rank you built up in social bookmarking sites or search engines will be lost. If you absolutely have to change a URL, ensure you set up a permanent 301 redirect to the new one so that your users don't get 404 errors. The W3C put it best: "Cool URLs don't change."

Only use disambiguated URLs: Any piece of content should have one and only one definitive URL, with any alternatives acting as a permanent redirect. In the past, features such as Apache's `DirectoryIndex` have meant that if you entered a URL that resolved to a folder, the default document for that folder would be served. This means that two URLs would exist for one resource. To make matters worse, servers are often configured so that `http://www.example.com/someresource` and `http://example.com/someresource` both point to the same resource. This means there can easily be four URLs for the same resource.

There are three good reasons why this is bad:

- Search engines and social bookmarking sites give pages with the most links the highest rank. If you have four different URLs for the same page, different people are likely to link to different versions, so you are effectively dividing your rank by 4.
- Browser or server caches will have to cache four versions of the page. Put another way, this means they can't improve performance if a user visits a different version of the same URL the second time.
- All versions of the page will be treated by web browsers as different resources, so the user's browsing history won't be accurate.

The only time you might want to have more than one URL for the same resource is if all but one of the URLs redirects to the other one so that users of your site who have already bookmarked links can still find them.

Treat the URL as part of the UI: Navigation links, sidebars, and tabs are all well and good, but if you have a good URL structure, your users should be able to navigate your site by changing parts of the URL. There are a few rules about how best to do this:

- Ensure that for every part of the path info part of a URL that a user might remove, a useful page is returned. For example, if the URL `/2007/nov/entry` gives a blog entry, `2007/nov` might list all the November entries, and `/2007` might list all entries from 2007.
- Never have a URL on your domain that gives a 500 error. It doesn't take a genius to realize that you don't want any URL in your web site to crash and cause a server error, but developers don't always think about what will happen if a user starts hacking the URL to contain different values. For example, if you have a URL `/food/apple` and a user changes it to `/food/pizza` when the application is set up only to deal with fruit, it should give a 404 Page Not Found error, not a 500 Internal Server error. Users are much more likely to stop trying to guess URLs if they get a 500 page because they'll be worried either that they might be breaking something or that the site is of low quality. The moment they stop hacking the URL, it has lost its usefulness as a UI component.

The most important tip is that you should use common sense when designing a URL structure. Don't apply any of the tips too rigidly; after all, you know your application and your users' requirements, so you can use your judgment about what will work best for you.

Unnecessary Routes Features

Because Routes was originally ported from the Ruby on Rails version, a few "features" have been added that in hindsight aren't necessary and are best to disable. In the following sections, you'll look at these features, what they do, and how to disable them.

■ **Note** It is highly likely that all these features will be completely removed in a future release of Routes.

Route Minimization

Previous versions of Pylons had a feature called *route minimization* enabled by default. Route minimization allowed you to specify optional default variables for a particular route. To demonstrate route minimization, let's look at the previous example again but this time with minimization enabled and with an optional default variable, `century`:

```
>>> map = Mapper(controller_scan=controller_scan)
>>> map.minimization = True
>>> map.connect('/archives/by_eon/{century}', controller='page',
...      action='aggregate', century=1800)
>>> print map.match('/archives/by_eon/')
{'action': u'aggregate', 'controller': u'page', 'century': u'1800'}
>>> print map.match('/archives/by_eon')
{'action': u'aggregate', 'controller': u'page', 'century': u'1800'}
>>> print map.match('/archives/by_eon/1800')
{'action': u'aggregate', 'controller': u'page', 'century': u'1800'}
```

Notice this time that all three URLs result in a match, even though two of them don't have a part of the URL that could correspond to the `century` dynamic part. This is because with minimization enabled, Routes assumes you want to use the optional default variables automatically if they aren't specified in the URL.

Here are some of the reasons why route minimization is a bad idea:

- It means multiple URLs serve the same page, which as explained earlier is bad for caches and search engine rankings.
- It makes it difficult to choose which URL should be generated for a particular action when using `h.url_for()`, so Routes always generates the shortest URL with minimization enabled.
- Complex routes and many defaults can make it hard to predict which route will be matched.

It is recommended that you leave route minimization disabled by keeping `map.minimization` set to `False` at the top of your route map.

Route Memory

One feature of Routes that was designed to make your life as a developer easier was called *route memory*. When your controller and action are matched from the URL, the routing variables that are set get remembered for the rest of that request. This means that when you are using `h.url_for()`, you actually only need to specify routing variables, which are different from the ones matched in the current request.

Consider this route:

```
map.connect('/archives/{year}/{month}/{day}', controller='page', action='view')
```

If the URL `/archives/2008/10/4` was entered, this route would match the following variables:

```
{'controller': 'page', 'action': 'view', 'year': '2008', 'month': '10', 'day': '4'}
```

With these routing variables matched, `h.url_for()` would generate the following results:

```
h.url_for(day=6)          # =>     /page/2008/10/6
h.url_for(month=4)        # =>     /page/2008/4/4
h.url_for(year=2009)      # =>     /page/2009/10/4
```

The majority of times you use `h.url_for()` will be in view templates. Since the same template is often called from multiple different controllers and actions, you can't always rely on `h.url_for()` generating the same URL in the same template, and this can introduce unexpected bugs, so it is usually much better to specify all your routing arguments explicitly.

There is another problem with route memory. Consider a route map set up with the following routes:

```
map.connect('/{controller}/{action}')
map.connect('/{controller}/{action}/{id}')
```

With route memory enabled, let's imagine someone visited the URL `/page/view/1`. Here are the routing variables that would be matched:

```
{'controller': 'page', 'action': 'view', 'id': '1'}
```

Now try to work out which URL would be generated with this code:

```
h.url_for(controller='page', action='new')
```

The result is `/page/new/1` even though the `new()` action might not take an `id` argument. This means that in order to have this matched by the first route to generate the URL `/page/new`, you would have to explicitly specify `id=None` like this:

```
h.url_for(controller='page', action='new', id=None)
```

You may not have noticed, but this is exactly what you did in the footer for the SimpleSite `view.html` template in Chapter 8. If route memory was disabled globally, you would not have had to specify `id=None` for the `new()`, `list()`, or `delete()` action routes.

As well as explicitly setting a routing variable to `None`, you can also disable route memory on a per-call basis by specifying the controller starting with a `/` character. For example:

```
h.url_for(controller='/page', action='new')
```

This is not ideal either, though. To avoid this complication, it is highly recommended that you disable route memory completely and always specify all arguments to `h.url_for()` explicitly. You can do this by passing `explicit=True` to the `Mapper()` constructor in your project's `config/routing.py`. This will ensure route memory isn't used, and you will have to explicitly specify each routing variable in every call to `h.url_for()`:

```
map = Mapper(directory=config['pylons.paths']['controllers'],
      always_scan=config['debug'], explicit=True)
```

If you should choose not to disable the route memory feature, you should be aware that it works only with `h.url_for()` and not with its counterpart `redirect_to()`.

Implicit Defaults

Routes has a surprising legacy feature that means that if you don't specify a controller and an action for a particular route, the *implicit defaults* of `controller='content'` and `action='index'` will be used for you. This can be demonstrated very simply:

```
>>> map = Mapper(controller_scan=controller_scan)
>>> map.minimization = False
>>> map.connect('/archives/')
>>> print map.match('/archives/')
{'action': u'index', 'controller': u'content'}
```

It is highly unlikely you want Routes to make up `action` and `controller` variables for you, so it's strongly recommended that you disable this feature. You can do this by passing `_explicit=True` as an argument to the individual routes you want implicit defaults to be disabled for:

```
>>> map = Mapper(controller_scan=controller_scan)
>>> map.minimization = False
>>> map.connect('/archives/', _explicit=True)
>>> print map.match('/archives/')
None
```

In reality, it is likely you will want these implicit defaults disabled for all your routes. Luckily, the `explicit=True` option you passed to the `Mapper()` object to disable route memory also disables implicit defaults, which is another reason to use it.

Best Practice

With route minimization, route memory, and implicit defaults disabled, the way you use Routes in your application might be quite different from the way you used it in previous versions of Pylons. To demonstrate this, let's look at an old routing structure that you might have used in Pylons 0.9.6:

```
>>> map = Mapper(controller_scan=controller_scan)
>>> map.minimize = True
>>> map.connect(':controller/:action/:id', controller='blog', action='view', id=1)
```

Here minimization is enabled as well as implicit routes and route memory. Look how many different URLs match the same controller, action, and ID. This clearly breaks the URL disambiguation rule listed earlier in the "Choosing Good URLs" section:

```
>>> print map.match('/blog/view/4/')
{'action': u'view', 'controller': u'blog', 'id': u'1'}
>>> print map.match('/blog/view/4')
{'action': u'view', 'controller': u'blog', 'id': u'1'}
>>> print map.match('/blog/view/')
{'action': u'view', 'controller': u'blog', 'id': u'1'}
>>> print map.match('/blog/view')
{'action': u'view', 'controller': u'blog', 'id': u'1'}
>>> print map.match('/blog/')
{'action': u'view', 'controller': u'blog', 'id': u'1'}
>>> print map.match('/blog')
{'action': u'view', 'controller': u'blog', 'id': u'1'}
>>> print map.match('/')
{'action': u'view', 'controller': u'blog', 'id': u'1'}
```

With route minimization, route memory, and implicit defaults disabled, you would have to add the following routes to achieve the same effect:

```
>>> map = Mapper(controller_scan=controller_scan, explicit=True)
>>> map.minimize = False
>>> map.connect('/', controller='blog', action='view', id=1)
>>> map.connect('/{controller}', action='view', id=1)
>>> map.connect('/{controller}/', action='view', id=1)
>>> map.connect('/{controller}/{action}', id=1)
>>> map.connect('/{controller}/{action}/', id=1)
>>> map.connect('/{controller}/{action}/{id}')
>>> map.connect('/{controller}/{action}/{id}/')
```

Of course, as has been mentioned, it is generally a bad idea to have more than one URL map to the same resource, so you are unlikely to want to do this in most applications. Even if you did want to do this, the explicit approach requires a lot more typing, but it is also much clearer what is happening. Although using explicit routes can sometimes mean more typing, being explicit about what you expect to happen makes life much easier when you have more complicated route setups.

Disabling route memory, implicit defaults, and route minimization also requires you to do more work when generating URLs, but once again, this will prevent you from finding obscure URL generation problems later in a project. To generate the URL to view the blog post with ID 1 and with the `explicit` argument set to `True`, you would always have to type the following, no matter which routing variables were generated for the request that calls the code:

```
h.url_for(controller='blog', action='view', id=1)
```

This would generate the URL `/` because this is the first route to match. Since routes are matched from top to bottom, if you wanted this to generate the URL `/blog/view/1` instead you could move the second-to-last route to the top so that it was matched first. This would affect the way URLs were generated for other controllers too, though:

```
>>> map = Mapper(controller_scan=controller_scan, explicit=True)
>>> map.minimize = False
>>> map.connect('/{controller}/{action}/{id}')
>>> map.connect('/', controller='blog', action='view', id=1)
>>> map.connect('/{controller}', action='view', id=1)
>>> map.connect('/{controller}/', action='view', id=1)
>>> map.connect('/{controller}/{action}', id=1)
>>> map.connect('/{controller}/{action}/', id=1)
>>> map.connect('/{controller}/{action}/{id}/')
```

Alternatively, you could add an *explicit route* as the first route. You'll remember from the discussion of default variables earlier in the chapter that an explicit route matches only *one* URL:

```
>>> map = Mapper(controller_scan=controller_scan, explicit=True)
>>> map.minimize = False
>>> map.connect('/blog/view/1', controller='blog', action='view', id=1)
>>> map.connect('/', controller='blog', action='view', id=1)
>>> map.connect('/{controller}', action='view', id=1)
>>> map.connect('/{controller}/', action='view', id=1)
>>> map.connect('/{controller}/{action}', id=1)
>>> map.connect('/{controller}/{action}/', id=1)
>>> map.connect('/{controller}/{action}/{id}')
>>> map.connect('/{controller}/{action}/{id}/')
```

Some people would argue that you should always specify the controller and action for *every* URL as hard-coded default variables using an explicit or named route. There are some other advantages to this approach:

- Temporarily blocking access to a particular controller can be difficult with nonexplicit routes because it isn't always obvious whether another route you have specified will also resolve the URL to the controller. If you have named or explicit routes, you can just comment out the routes that point at the controller in question, and you're all set.
- Imagine you have a Pylons application that is a couple of years old and has an established group of users who have bookmarked various parts of your site. If you were to refactor the application, moving controllers around or putting certain actions in different controllers, you would find it difficult to maintain the existing URL structure if you were using implicit routes. With explicit or named routes, you could simply update where the URL pointed to so that the old URLs would still work.

I hope what is clear from the discussion in this section is that you should always set `explicit=True` in your `Mapper` to disable route memory and implicit defaults and that you should disable route minimization with `map.minimization=False`, but it is up to you whether you choose to go further and be even more explicit about your routes with hard-coded defaults for the controller and action or named routes.

Only you can decide whether ordinary, explicit, or named routes are the correct choice for your application. Ordinary routes that use the "convention over configuration" philosophy don't result in disambiguated URLs. Named and explicit routes give you complete control over how your URLs are generated but are more effort to set up. The nice thing about Routes is, of course, that you can use all three techniques together.

Advanced URL Routing

It is now time to learn about some of the really advanced features of Routes that, when used effectively, can very much simplify your route maps or enable your Pylons application to deal with any number of complex legacy URL structures you might have inherited from an old application.

Requirements

Any route you specify can have a `requirement` argument specified as a keyword to the `map`'s `connect()` method. The requirement argument should be a dictionary of routing variable names for any dynamic or wildcard parts and a corresponding regular expression to match their values against. Regular expressions are documented in the Python documentation for the `re` module. Any valid Python regular expression can be used.

Here's an example:

```
map.connect('archives/{year}/{month}/{day}', controller='archives', action='view',
    year=2004,requirements={'year': '\d{2,4}', 'month': '\d{1,2}'})
```

One particularly useful regular expression used in the previous example is `\d`, which matches a digit. In this case, the year can be either two or four digits long, and the month can be one or two digits long.

Here's another example; this time we are ensuring the theme is one of the allowed themes:

```
theme_map = 'admin|home|members|system'
map.connect('users/{theme}/edit', controller='users',
    requirements={'theme': theme_map})
```

Conditions

Conditions specify a set of special conditions that must be met for the route to be accepted as a valid match for the URL. The `conditions` argument must always be a dictionary and can accept three different keys:

`method`: The request must be one of the HTTP methods defined here. This argument must be a list of HTTP methods and should be uppercase.

`function`: This is a function that will be used to evaluate whether the route is a match. This must return `True` or `False` and will be called with `environ` and `match_dict` as arguments. The `match_dict` is a dictionary with all the routing variables for the request. Any modifications your function makes to the `match_dict` will be picked up by Routes and used as the routing variables from that point on.

`sub_domain`: If this is present, it should be either `True` (which will match any subdomain) or a Python list of subdomain strings, one of which must match the subdomain used in the request for the route to match.

These three types of conditions can all be used together in the same route by specifying each argument to the `conditions` dictionary, but I'll discuss them each in turn.

Let's deal with the `method` option first. This allows you to match a URL based on the HTTP method the request was made with. One problem with testing examples using the `method` condition is that the match depends on information from the WSGI environment, which is usually set during a request. Since there isn't a real HTTP request for this test code, you have to emulate it, so in the following examples you set `map.environ` with fake environment information to test how the matching works. Obviously, you wouldn't do this in your own code. The example also sets up a new `controller_scan()` function which only accepts the value `user` for the controller:

```
>>> from routes import Mapper
>>> def controller_scan(directory=None):
>>>     return ['user']
...
>>> map = Mapper(controller_scan=controller_scan)
>>> map.minimization = False
>>> map.connect('/user/new/preview', controller='user', action='preview',
...      conditions=dict(method=['POST']))
>>> # The method to be either GET or HEAD
>>> map.connect('/user/list', controller='user', action='list',
...     conditions=dict(method=['GET', 'HEAD']))
>>> map.environ = {'REQUEST_METHOD':'POST'}
>>> print map.match('/user/new/preview')
{'action': u'preview', 'controller': u'user'}
>>> print map.match('/user/list')
None
>>> map.environ = {'REQUEST_METHOD':'GET'}
>>> print map.match('/user/new/preview')
None
>>> print map.match('/user/list')
{'action': u'list', 'controller': u'user'}
```

As you can see, the `method` condition works as expected, allowing the routes to match only when the request contains the correct request method.

Now let's look at the `function` condition. This condition is extremely powerful because it effectively allows you to write custom code to extend Routes' functionality. The function can return `True` to indicate the route matches or `False` if it doesn't. The function also has full access to the WSGI

environment and the dictionary of routing variables, which have already been matched, so the function is free to modify or add to them as well.

Here's an example of a function that extracts the action to be called from the `X-ACTION` HTTP header in the request and compares it to allowed values:

```
from webob import Request
def get_action(environ, result):

    # Create a Pylons-style request object from the environment for
    # easier manipulation
    req = Request(environ)

    action = req.GET.get('X-ACTION')
    if action in ['call', 'get', 'view']:
        result['action'] = action
        return True

    return False
```

You wouldn't often design this sort of functionality into your routes if you were creating your own application from scratch, but having such low-level access to the underlying details of the request is very powerful and can be useful when writing a Pylons application to replace legacy code.

Here is another use that might be useful in your application, this time to treat the referrer as an ordinary routing variable:

```
def referals(environ, result):
    result['referer'] = environ.get('HTTP_REFERER')
    return True
```

You could use this function in a route like this to add the referrer to the matched routing variables:

```
map.connect('/{controller}/{action}/{id}', conditions=dict(function=referals))
```

Now let's think about subdomains. These are easiest to demonstrate with examples, and once again you have to create a fake HTTP request environment for your examples to emulate requests coming from different subdomains.

First let's set up the `map` object. Notice that you specify `map.sub_domains=True` to enable subdomain support:

```
>>> from routes import Mapper
>>> def controller_scan(directory=None):
...     return ['user']
...
>>> map = Mapper(controller_scan=controller_scan)
>>> map.minimization = False
>>> map.sub_domains = True
```

Now let's set up two routes. The first will accept any subdomain and produce an `action` routing variable with a value `any`, and the second will accept only the subdomains `foo` and `bar` and will produce an `action` routing variable `certain`:

```
>>> map.connect('/user/any', controller='user', action='any',
... conditions={'sub_domain':True})
>>> map.connect('/user/certain', controller='user', action='certain',
... conditions={'sub_domain': ['foo', 'bar']})
```

Now let's attempt some matches, emulating different requests by setting different `environ` dictionaries. First let's emulate a domain `foo.example.com` and test two URLs. The first will test what happens when `sub_domain` is set to `True`, and the second will test what happens when it is restricted to `['foo', 'bar']`:

```
>>> map.environ = {'HTTP_HOST':'foo.example.com'}
>>> print map.match('/user/any')
{'action': u'any', 'controller': u'user', 'sub_domain': 'foo'}
>>> print map.match('/user/certain')
{'action': u'certain', 'controller': u'user', 'sub_domain': 'foo'}
```

As you can see, both the routes match because `foo` is a valid subdomain for either condition. Notice how `'sub_domain': 'foo'` is added to the matched routing variables.

Now let's try the domain `not.example.com`:

```
>>> map.environ = {'HTTP_HOST':'not.example.com'}
>>> print map.match('/user/any')
{'action': u'any', 'controller': u'user', 'sub_domain': 'not'}
>>> print map.match('/user/certain')
None
```

This time, since `not` is not one of the allowed domains for the second route, only the first example works. Let's see what happens if no subdomain is specified in the request:

```
>>> map.environ = {'HTTP_HOST':'example.com'}
>>> print map.match('/user/any')
None
>>> print map.match('/user/certain')
None
```

As you can see, neither of the routes matches.

Caution To be able to use matching based on subdomain, you must set the `map`'s `sub_domains` attribute to `True`. Otherwise, none of the routes checking the subdomain conditions will match.

To avoid matching common aliases to your main domain like `www`, the subdomain support can be set to ignore certain specified subdomains. Here's an example:

```
>>> from routes import Mapper
>>> def controller_scan(directory=None):
...     return ['user']
...
>>> map = Mapper(controller_scan=controller_scan)
>>> map.minimization = False
>>> # Remember to turn on sub-domain support
... map.sub_domains = True
>>> # Ignore the ``www`` sub-domain
... map.sub_domains_ignore = ['www']
>>> map.connect('/user/any', controller='user', action='any',
...     conditions=dict(sub_domain=True))
>>> map.connect('/user/certain', controller='user', action='certain',
...     conditions=dict(sub_domain=['www', 'foo']))
>>> map.environ = {'HTTP_HOST':'foo.example.com'}
>>> print map.match('/user/any')
{'action': u'any', 'controller': u'user', 'sub_domain': 'foo'}
```

```
>>> print map.match('/user/certain')
{'action': u'certain', 'controller': u'user', 'sub_domain': 'foo'}
>>> map.environ = {'HTTP_HOST':'www.example.com'}
>>> print map.match('/user/any')
None
>>> print map.match('/user/certain')
None
```

You can see that requests to the `www` subdomain are not matched even if one of the conditions specifies that the subdomain `www` should be matched. Now let's look at how `h.url_for()` handles subdomains.

When subdomain support is on, the `h.url_for()` function will accept a `sub_domain` keyword argument. Routes then ensures that the generated URL has the subdomain indicated. For example:

```
h.url_for(controller='user', action='certain', sub_domain='foo')
```

would generate the following URL but it will only work during a Pylons request, not from the interactive Python session we've been using so far in this chapter:

```
http://foo.example.com/user/certain
```

Filter Functions

Named routes can have functions associated with them that will operate on the arguments used during generation. Filter functions don't work with implicit or explicit routes because the filter function itself can affect the route that actually gets chosen, so the only way to be explicit about which route to use is to specify it with a name.

To highlight the problem filter functions solve, consider this route:

```
map.connect('/archives/{year}/{month}/{day}', controller='archives',
          action='view', year=2008,
          requirements=dict(year='\d{2,4}', month='\d{1,2}'))
```

Generating a URL for this will require a month and day argument and a year argument if you don't want to use 2008. Imagine this route links to a story and that your model has a story object with `year`, `month`, and `day` attributes. You could generate a URL for the story like this:

```
h.url_for(year=story.year, month=story.month, day=story.day)
```

This isn't terribly convenient and can be brittle if for some reason you need to change the `story` object's interface. It would be useful to be able to pass the `story` object directly to `h.url_for()` and have Routes extract the information it requires automatically. You can do this with a filter function.

Here's what the filter function for the `story` object might look like:

```
def story_expand(result):
    # Only alter args if a story keyword arg is present
    if 'story' not in result:
        return result

    story = result.pop('story')
    result['year'] = story.year
    result['month'] = story.month
    result['day'] = story.day

    return result
```

You can then create a named route with a `_filter` argument like this:

```
m.connect('archives', '/archives/{year}/{month}/{day}',
          controller='archives', action='view', year=2004,
          requirements=dict(year='\d{2,4}', month='\d{1,2}'),
          _filter=story_expand)
```

This filter function will be used when using the named route `archives`. If a `story` keyword argument is present, it will use that and alter the keyword arguments used to generate the actual route.

If you have a `story` object with those attributes, you would now be able to generate a URL like this:

```
h.url_for('archives', story=my_story)
```

As well as making it substantially easier to generate the URL, you can also easily change how the arguments are pulled out of the `story` object simply by changing the filter function if the `story` object interface were ever to change.

Although filter functions can be very powerful, you might decide against using them in your application because they can also make it less obvious what is happening to generate your URLs.

Summary

You've now seen all the main features of Routes and how you can take advantage of them to perform some very complex URL mappings as simply as possible or to integrate with legacy systems. You also saw how some of Routes legacy features such as route minimization, route memory, and implicit defaults can cause problems you might not have expected, and you now know how to avoid using these features in your application. Although route minimization is disabled by default in new Pylons applications, route memory and implicit defaults are not, so it is recommended you set `explicit=True` as an argument to `Mapper()` in your project's `config/routing.py` file.

In Chapter 14, you'll look at Routes again to see how you can use its features to improve the SimpleSite application.