CHAPTER 16

The Web Server Gateway Interface (WSGI)

The Web Server Gateway Interface is a Python standard created in 2003 by Philip J. Eby and the Python web community. Back in 2003 Python suffered from a very fragmented web framework community where applications written with code from one framework wouldn't run on the server component from a different framework. The Web Server Gateway Interface standard (known as WSGI and pronounced "wizgy" by those in the Python community) was designed to change that and enable a degree of cross-framework interoperability.

In the first part of this chapter, you'll learn how a simple WSGI application works, how you can use WSGI applications as Pylons controllers, how WSGI servers work, and how WSGI leads to new classes of components called *middleware*.

Internally Pylons relies on middleware components to provide some of its core functionality, so in the second part of the chapter, I'll explain the different ways of writing WSGI middleware, and you'll develop your own Gzip middleware (mentioned in the previous chapter) for compressing your Pylons projects' CSS and JavaScript code.

In the normal course of Pylons development, you don't need to write WSGI applications because you will use Pylons controllers and actions instead. You don't need to know how to develop WSGI middleware yourself either because Pylons already provides all the middleware you need as well as specific APIs that are much easier to work with than their underlying WSGI APIs (for example, those exposed by the `request` and `response` objects). For these reasons, you might want to skip the "Writing WSGI Middleware" section in the second half of the chapter if it doesn't apply to you yet and move straight onto the next chapter at that point. You should still read "Introducing WSGI" in this chapter, though.

Introducing WSGI

The WSGI standard defines three different classes of component: *applications*, *servers*, and *middleware*. I'll cover each of these types of components in turn and explain how they are relevant to a Pylons application.

Tip If you want to know the details of WSGI programming, there is no substitute for reading its formal specification, which is called Python Enhancement Proposal 333 and is at `http://www.python.org/dev/peps/pep-0333/`.

WSGI Applications

You'll remember from all the way back in Chapter 3 that at its heart Pylons deals with the HTTP protocol, that communication with the browser involves a request and a response, and that all the HTTP request information, together with information about the server, is encapsulated in a CGI-like environment dictionary.

In Pylons, a controller action to handle a request and return some plain text might look like this:

```
def hello(self):
    response.status = '200 OK'
    response.content_type = 'text/plain'
    return "Hello World!"
```

The equivalent WSGI application would look like this:

```
def hello(environ, start_response):
    start_response('200 OK', [('Content-type','text/plain')])
    return ["Hello World!"]
```

Technically speaking, setting the response status to `'200 OK'` in the Pylons example isn't necessary because it is the default, but I've added it to the example to emphasize the similarity between the two cases. Let's compare each of the examples by their main features:

Request information: In Pylons, all the request information is available as a global object called `request`. In the WSGI application, the request information is all contained in the `environ` dictionary passed as the first positional parameter to the WSGI application. In Pylons, `request.environ` is actually the same dictionary that would be passed to a WSGI application as the `environ` argument.

HTTP response status: In Pylons, the status is set by changing the `status` attribute of the global `response` object. In a WSGI application, it is set by passing a string as the first argument to the `start_response()` function, which itself is passed into the WSGI application as its second positional parameter.

HTTP headers: In Pylons, common HTTP headers such as `Content-Type` have their own attributes that you can set on the global `response` object. Others have to be added to `response.headers`, as in `response.headers['X-Some-Header'] = 'value'`. In a WSGI application, the headers are passed as the second argument to `start_response()` as a list of tuples where the first string in a tuple is a string containing the header name and the second is a string containing the value. Here's an example: `[('Content-type', 'text/plain'), ('X-Some-Header', 'value')]`.

The response: In Pylons, the response is simply the string returned from the action. If you return Unicode from a Pylons controller action, it will be encoded into UTF-8 automatically. In a WSGI application, it is an iterable, which should yield strings when it is iterated over. A list made up of strings is an example of an iterable that fulfils this criterion. The WSGI response cannot contain Unicode. Any Unicode must be encoded to an encoding such as UTF-8 before it is used in an iterable.

On the surface, the WSGI application is really fairly similar to the Pylons controller action except it doesn't use the simplifying global variables `request` and `response` that Pylons provides; however, you need to be aware of a couple of complications when writing WSGI applications, which Pylons takes care of for you when you use a controller action:

- The `start_response()` callable that gets passed to the WSGI application as its second argument can be called only *once* (except in rare circumstances when an error occurs, as you'll see in the "Handling Errors" section later in the chapter).
- `start_response()` *must* be called *before* the application returns any response data.
- After being called, `start_response()` returns a writable object that can be used to write response data directly without having to return it as an iterable from a WSGI application.

Here is the same example you saw earlier but written to use the writable object returned from `start_response()`:

```
def hello(environ, start_response):
    writable = start_response('200 OK', [('Content-type','text/html')])
    writable("Hello ")
    return ["World!"]
```

Here the first part of the output was written via the writable returned by `start_response()`, and the rest was returned normally by returning the iterable as before.

■ **Caution** This way of returning response data to the browser was included in the specification because some servers weren't capable of buffering data returned in any other way. It is really considered very bad practice to use this approach; in fact, this functionality may be removed in a future version of the WSGI specification because it significantly complicates the writing of middleware components.

Using Instances of Classes

So far, you've seen only how to write WSGI applications as functions, but they can also be written as iterators, as generators, or as class instances. Writing applications as class instances can be useful if you want to write a more complex WSGI application. Take a look at this example:

```
class Hello(object):

    def __call__(self, environ, start_response):
        start_response('200 OK', [('Content-type','text/plain')])
        return ['Hello World!']

hello = Hello()
```

The `Hello` class itself isn't a WSGI application, but if you think about how the `hello` instance will behave, you'll realize that when it is called, it accepts two positional parameters; it calls `start_response()` and returns an iterable, which in this case is just a list with one string. These are exactly the conditions outlined in the previous section to describe how a WSGI application should behave, so the class *instance* is a valid WSGI application. This approach is very handy for two main reasons:

- It provides a way to group related WSGI applications together by having the `__call__()` dispatch the request to different methods.
- It provides a way to configure an application by passing arguments to the `__init__()` method.

Let's see an example demonstrating these two uses:

```
class Application(object):
    def __init__(self, name):
        self.name = name

    def __call__(self, environ, start_response):
        if environ['PATH_INFO'] == '/hello':
            return self.hello(environ, start_response)
        elif environ['PATH_INFO'] == '/goodbye':
            return self.goodbye(environ, start_response)
        else:
            start_response('404 Not Found', [('Content-type','text/html')])
            return ['Not found']

    def hello(self, environ, start_response):
        start_response('200 OK', [('Content-type','text/html')])
        return ['Hello %s'%(self.name)]

    def goodbye(self, environ, start_response):
        start_response('200 OK', [('Content-type','text/html')])
        return ['Goodbye %s'%(self.name)]

app = Application('Harry')
```

In this example, you can see that the method that is executed depends on the path in the URL and that the text returned from the two methods depends on the name that was given when the WSGI application was created. You might notice that this is beginning to look like an ordinary Pylons controller, and as you'll find out in the next section, it turns out that Pylons is designed to be able to run WSGI applications like this as Pylons controllers.

■ **Caution** It is easy to fall into a trap when using class instances in this way as WSGI applications. You might be tempted to pass variables between the class methods by assigning values to `self` and then calling other methods. This would be fine if you were using the application only outside of a multithreaded environment or if you re-created the application on each request, but in a multithreaded environment there would be no guarantee that the value attached to `self` in one method call would be the same value that was used in the next method call, because another thread of execution might have taken place in between.

If that sounds a bit confusing, just remember the simple rule that you should never set or change a variable assigned to `self` in a WSGI application outside the `__init__()` method.

This isn't a problem in a Pylons controller because Pylons controller instances are re-created on each request.

WSGI in Pylons Controllers

If you look at a Pylons controller, you will see it is derived from `BaseController`, which is defined in your project's `lib/base.py` file. The `BaseController` class is itself derived from `pylons.controllers.WSGIController`, but in fact Pylons is designed so that any valid WSGI application can be used as a controller. This means you don't actually need to use a Pylons controller class in your controller at all; any WSGI application will work as long as you give it the same name your controller would have had so that Pylons can find it.

For example, if you added a `hello` controller to a Pylons application by running the `paster controller hello` command, you could replace the entire contents of the `hello.py` file with this:

```
def HelloController(environ, start_response):
    start_response('200 OK', [('Content-Type','text/plain')])
    return ['Hello World!']
```

Pylons will call your custom WSGI application in the same way as it would call a normal `WSGIController` instance. To test the example you'll need to visit a URL which will resolve to the controller such as `/hello/index` or add a new route so that the controller can be accessed as `/hello`:

```
map.connect('/hello', controller='hello')
```

You will still have all the session, debugging, and error-handling facilities a normal Pylons controller has. The only difference is that although Pylons would instantiate a new `WSGIController` on each request, it will assume that any WSGI application you use either is a function (such as the `HelloController()` function earlier) or is already instantiated and ready to handle multiple requests at once. This means if you want to write a WSGI application as a class and want it to be used as a controller, the class *instance* would have to be named `HelloController`, not the class itself. Here's an example:

```
class HelloControllerApplication:
    def __call__(self, environ, start_response):
        start_response('200 OK', [('Content-Type','text/plain')])
        return ['Hello World!']

HelloController = HelloControllerApplication()
```

Pylons avoids this problem with controllers derived from `pylons.controllers.WSGIController` because it automatically creates a new controller instance on each request before it calls it.

Being able to mount WSGI applications as Pylons controllers is very useful because it gives you the basis for integrating Pylons applications with third-party WSGI-enabled software such as MoinMoin or Mercurial. As an example, it is perfectly possible to mount an entire Trac instance as a Pylons controller. This has the benefit that Trac will be able to use the same authentication and authorization system that you are using with Pylons and will also benefit from Pylons' automatic error documents and interactive debugger.

There may be occasions where you don't want to replace your entire controller with a WSGI application but simply want to run a WSGI application from within a particular controller action. For example, if you had a WSGI application called `wsgi_app`, you could call it from the `index()` controller action like this:

```
# WSGI application
def wsgi_app(environ, start_response):
    start_response('200 OK',[('Content-type','text/html')])
    return ['<html>\n<body>\nHello World!\n</body>\n</html>']

# Pylons controller
class RunwsgiController(BaseController):

    def index(self, environ, start_response):
        return wsgi_app(environ, start_response)
```

Notice that the WSGI objects `environ` and `start_response` are automatically passed to the Pylons controller action if it has their names as arguments. This is another feature of the Pylons dispatch designed to make working with WSGI applications easier, but you'll remember from Chapter 9 that this is why you shouldn't choose routing variables with the same names.

Not all WSGI applications can be run as or from Pylons controllers. Some of the problems are as follows:

- Python cannot support more than one version of the same library in the same interpreter at the same time, so the WSGI application cannot use a different version of one of the libraries Pylons is already using.
- Not all WSGI applications are written correctly to be mounted at a URL other than `/`, so when you use them as part of a Pylons controller at a URL such as `/controller/action`, they might break.
- Not all WSGI applications are compatible with the WSGI middleware components Pylons uses. For example, they might not expect the presence of error documents middleware that could get in the way of some of their responses.

Despite these potential difficulties, a lot of WSGI applications will run from within Pylons, and this can lead to interesting new ways of using those applications.

WSGI Servers

Now that you've seen what WSGI applications are and how, in many cases, they can be run from a Pylons controller, it is time to turn your attention to what happens with WSGI servers. Luckily, you are very unlikely to have to write a WSGI server yourself because WSGI is now an established standard and almost all the common web servers are now WSGI compatible. Nevertheless, it is still useful to know how they work since their interface forms the basis for WSGI middleware.

A WSGI server must do a few things. First, it must prepare the `environ` dictionary and `start_response()` callable and pass them as the first and second positional parameters to a WSGI application. The WSGI application always calls the `start_response()` callable to set the status and HTTP headers before it starts returning data (as you've seen), so the server has to assemble `start_response()` in such a way that, when it is called, the status and headers get sent to the web browser. You'll remember that according to the specification, WSGI applications are allowed to use the writable returned from `start_response()`, so servers must also assemble `start_response()` to return a compatible object.

It is also the server's responsibility to set certain WSGI-specific variables in the `environ` dictionary that give the application some information about the type of server they are running on. Table 16-1 lists the variables that are set.

Table 16-1. *The WSGI Variables and Their Meanings According to PEP 333*

Variable	Value
`wsgi.version`	The tuple `(1,0)`, representing WSGI version 1.0.
`wsgi.url_scheme`	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`	An input stream (file-like object) from which the HTTP request body can be read. (The server or gateway may perform reads on demand as requested by the application, it may preread the client's request body and buffer it in memory or on disk, or it may use any other technique for providing such an input stream, according to its preference.)

Variable	Value
`wsgi.errors`	An output stream (file-like object) to which error output can be written, for the purpose of recording program or other errors in a standardized and possibly centralized location. This should be a "text mode" stream; that is, applications should use `"\n"` as a line ending and assume it will be converted to the correct line ending by the server/gateway. For many servers, `wsgi.errors` will be the server's main error log. Alternatively, this may be `sys.stderr` or a log file of some sort. The server's documentation should include an explanation of how to configure this or where to find the recorded output. A server or gateway may supply different error streams to different applications, if this is desired.
`wsgi.multithread`	This value should evaluate `true` if the application object may be simultaneously invoked by another thread in the same process and should evaluate `false` otherwise.
`wsgi.multiprocess`	This value should evaluate `true` if an equivalent application object may be simultaneously invoked by another process and should evaluate `false` otherwise.
`wsgi.run_once`	This value should evaluate `true` if the server or gateway expects (but does not guarantee!) that the application will be invoked only this one time during the life of its containing process. Normally, this will be `true` only for a gateway based on CGI (or something similar).

Of particular interest to a Pylons developer are the `wsgi.multithread` and `wsgi.multiprocess` variables that tell the WSGI application whether it is being run in a multithreaded or multiprocess environment. You'll learn more about the differences between multithreading and mulitprocess servers in Chapter 21, but for now you just need to know that these variables can be accessed from the `environ` dictionary to discover what sort of server is being used.

The `wsgi.errors` variable is also interesting and will become particularly relevant when you look at application logging in Chapter 20. It provides a file-like object that the server provides for you to log error messages to. By using this for your log messages, you can be sure that your Pylons application logs will always be logged in an appropriate manner no matter which server they are running on. There are downsides to this approach too, though, which you'll see in Chapter 20.

■ **Tip** Because Pylons is a WSGI framework and runs on WSGI servers, these variables are also available in Pylons through the `request.environ` dictionary. For example, to find out whether your application is running on a multi-threaded web server, you could do this:

```
if bool(request.environ['wsgi.multithreaded']):
    # Multithreaded
else:
    # Not multithreaded
```

Rather than write a full WSGI HTTP server as an example, let's consider a much simpler case. Imagine you already have a CGI server that was capable of running the CGI script you saw all the way back in Chapter 1. If you wanted to write an adaptor that would be able to run a WSGI application in a CGI environment, you would need to implement the WSGI server API.

Here's the code you would use:

```
import os, sys

def run_with_cgi(application):

    environ = dict(os.environ.items())
    environ['wsgi.input']        = sys.stdin
    environ['wsgi.errors']       = sys.stderr
    environ['wsgi.version']      = (1,0)
    environ['wsgi.multithread']  = False
    environ['wsgi.multiprocess'] = True
    environ['wsgi.run_once']    = True

    if environ.get('HTTPS','off') in ('on','1'):
        environ['wsgi.url_scheme'] = 'https'
    else:
        environ['wsgi.url_scheme'] = 'http'

    headers_set = []
    headers_sent = []

    def write(data):
        if not headers_set:
             raise AssertionError("write() before start_response()")

        elif not headers_sent:
             # Before the first output, send the stored headers
             status, response_headers = headers_sent[:] = headers_set
             sys.stdout.write('Status: %s\r\n' % status)
             for header in response_headers:
                 sys.stdout.write('%s: %s\r\n' % header)
             sys.stdout.write('\r\n')

        sys.stdout.write(data)
        sys.stdout.flush()

    def start_response(status,response_headers,exc_info=None):
        if exc_info:
            try:
                if headers_sent:
                    # Re-raise original exception if headers sent
                    raise exc_info[0], exc_info[1], exc_info[2]
            finally:
                exc_info = None     # avoid dangling circular ref
        elif headers_set:
            raise AssertionError("Headers already set!")

        headers_set[:] = [status,response_headers]
        return write
    result = application(environ, start_response)
    try:
        for data in result:
            if data:    # don't send headers until body appears
                write(data)
        if not headers_sent:
            write('')   # send headers now if body was empty
    finally:
        if hasattr(result,'close'):
            result.close()
```

This example forms part of the WSGI specification in PEP 333, so I won't discuss it in detail, but notice that it sets up an `environ` dictionary and a `start_response()` callable before calling the WSGI application in the line marked in bold. The information returned from the WSGI application forms the body of the response, and the information passed to the `start_response()` callable from the application is used to set the HTTP status and headers. Notice that the example also sets up a `write()` function that is returned from `start_response()` and that the application can also use to output information. As noted earlier, though, the vast majority of WSGI applications should return their data as an iterable. It's also worth drawing your attention to `wsgi.errors`. Here the `wsgi.errors` key simply points to `sys.stderr`, so any log messages your application sends when using `run_with_cgi()` just get sent to the standard error stream.

Let's write a sample CGI script that uses `run_with_cgi()` to run the `hello()` WSGI application from earlier in the chapter. The script would look like the following, but be sure to specify the correct path to the `python` executable. This is likely to be the one in your virtual Python environment:

```
#!/home/james/env/bin/python

import os, sys

def run_with_cgi(application):
    ... same as in the example above ...

def hello(environ, start_response):
    start_response('200 OK', [('Content-type','text/plain')])
    return ["Hello World!"]

if __name__ == '__main__':
    run_with_cgi(hello)
```

You can also run this CGI script from the command line because CGI applications send their output to the standard output. Save the previous example as `wsgi_test.py`, and you can run it like this:

```
$ chmod 755 wsgi_test.py
$ ./wsgi_test.py
Status: 200 OK
Content-type: text/plain

Hello World!
```

This sort of setup can sometimes be useful for debugging problems too because you can customize how the `environ` dictionary is set up in `run_with_cgi()` to simulate different sorts of requests.

Caution Running a WSGI application through a CGI script is generally very inefficient because the WSGI application (along with the Python interpreter and the CGI script itself) has to be re-created on each request. Most WSGI applications (including your Pylons application) are designed to be loaded into memory once and then executed lots of times without being re-created, and this is a lot more efficient.

Since Python 2.5, WSGI has been built into the Python standard library in the form of the `wsgiref` module, which provides basic WSGI tools including a WSGI server. The server is built using the same methodology as the other servers that make up the Python Standard Library including `BaseHTTPServer`. Here's how you would use it to serve the same `hello` WSGI application used in the previous example, although you could use the same code to serve *any* WSGI application just by changing the argument to `httpd.set_app()`:

```
from wsgiref import simple_server

def hello(environ, start_response):
    start_response('200 OK', [('Content-type','text/plain')])
    return ["Hello World!"]

httpd = simple_server.WSGIServer(
    ('0.0.0.0', 8000),
    simple_server.WSGIRequestHandler,
)
httpd.set_app(hello)
httpd.serve_forever()
```

Tip If you are running a version of Python prior to 2.5, you will need to install the `wsgiref` package from the Python Package Index to run the example. You can do that with the following:

```
$ easy_install wsgiref
```

If you run this application, you'll find the server running on port 8000 and available on all IP addresses. You change which interface or port the WSGI application is served from by changing the first argument to `simple_server.WSGIServer()`. Once the server is running, you can visit `http://localhost:8000/` to see the `hello` application running.

Now you have seen how to write WSGI applications and servers and understand the API for each, it should be clear that any WSGI application can run on any WSGI server without any modification needed to either the server or the application. In the next chapter, you'll learn how to obtain a WSGI application object from a Pylons application through its config file. Because Pylons applications are also WSGI applications, it means they can be deployed on a large range of WSGI servers without modification. You can serve these Pylons WSGI applications in the same ways you saw the `hello` application being served in the examples in this chapter. You'd just swap `hello` for the Pylons WSGI application. You'll learn about some of the more common deployment setups in Chapter 21.

WSGI Middleware

Now that you've learned about WSGI applications and servers, it is time to learn about another type of component that sits in the middle between a server and an application and is known as *middleware*.

WSGI middleware are components that, from a WSGI application's point of view, appear as though they are a WSGI server because they provide the `environ` dictionary and `start_response()` callable when they call the WSGI application and iterate over the result to return the response in the same way a WSGI server would. The reason middleware components are not WSGI servers, though, is that they look to a server as if they are a WSGI application. They are callables that accept the `environ` dictionary and `start_response()` arguments and return an iterable response.

This dual nature puts middleware components in a unique position to be able to change all the HTTP information an application receives from a server and to change all the HTTP information a server receives from the application. This turns out to be extremely useful and means that middleware can therefore do any of the following things or a combination of them:

- Change any of the request information by modifying the `environ` dictionary
- Change the HTTP status returned from an application
- Intercept an error
- Add, remove, or change HTTP headers returned from an application
- Change a response

Middleware is therefore extremely powerful and can build a broad range of discrete components that can be used with different WSGI servers and applications. For example, a middleware component can do the following:

- Produce error documents when certain status codes are received (typically responding to 404 and 500 codes)
- E-mail error reports to a developer if a problem occurs
- Provide interactive debugging facilities
- Forward requests to other parts of the application
- Test the API compliance of applications and servers to the WSGI standard
- Authenticate a user
- Cache pages
- Provide a session store
- Handle cookies
- Gzip the response

Since a WSGI application wrapped in a piece of WSGI middleware is still a valid WSGI application, you can also wrap the combined middleware+application in another piece of middleware. You can keep adding middleware components until your middleware stack provides all the functionality you need. Doing so is called creating a *middleware chain*. This is exactly what is happening in your Pylons application's `config/middleware.py` file in the `make_app()` function that you'll look at in detail in the next chapter.

Writing WSGI Middleware

Now that you've seen a bit about how WSGI works and how middleware components in particular can be used to change the behavior of a Pylons application, it is time to look in detail at how you can write WSGI middleware yourself. As was noted in the introduction to the chapter, writing your own middleware isn't necessary to develop Pylons applications, so feel free to skip the rest of this chapter if it doesn't interest you at this stage; however, bear in mind that if you have an understanding of how it is done, you will be better able to understand how components such as AuthKit work. If you want to become a Pylons expert or want to contribute to Pylons itself, a good understanding of WSGI applications and middleware is essential.

Let's start off by looking at middleware that does nothing at all so that you can see its constituent parts:

```
class Middleware(object):
    def __init__(self, app):
        self.app = app

    def __call__(self, environ, start_response):
        return self.app(environ, start_response)
```

You can use this middleware like this to wrap the `hello` WSGI application you saw earlier:

```
app = Middleware(hello)
```

The combined `app` object is itself a valid WSGI application, and since the middleware doesn't do anything, it would behave in the same way as the `hello` application on its own. Let's think about what happens when it is called by a WSGI server to handle a request.

When the `app` object is created, the `Middleware` object is instantiated with the `hello` application as its first argument, which gets set as `self.app`. The `app` object is a class instance, but since it has a `__call__()` method, it can be called and therefore can behave as a WSGI application in a similar manner to the way you saw a class instance being used as a WSGI application earlier in the chapter.

When the instance is called, it in turn calls the `hello` application (`self.app`) and then returns its result to the server, which will iterate over the result from `Middleware.__call__()` in the same way as it would have iterated over the result from `hello()` if `Middleware` wasn't present.

Now that you've seen the basic API of a middleware component, let's look at an example of each of the things mentioned in the previous section, which you can do with a middleware component starting with modifying the environment.

Modifying the Environment

Let's write some middleware that modifies the `environ` dictionary to add a key specifying a message. You'll also add a facility allowing the message to be configured when the middleware is instantiated:

```
class Middleware(object):
    def __init__(self, app, message):
        self.app = app
        self.message = message

    def __call__(self, environ, start_response):
        environ['example.message'] = self.message
        return self.app(environ, start_response)
```

This middleware adds a key to the `environ` called `example.message`. All WSGI `environ` keys have to be strings containing just one `.` character, so here, `example.message` is a valid key. A WSGI application can now access this modified example. Here's a new version of the `hello` application that uses this key to display a message and an example of how it is used:

```
def custom_message(environ, start_response):
    start_response('200 OK', [('Content-type', 'text/plain')])
    return [environ['example.message']]

app = Middleware(custom_message, "Hello world again!")
```

This time, the Middleware class takes a message parameter that is set as self.message. When the application is called, the middleware adds this message as a key in the environ dictionary, which the application can now extract from the environment.

Changing the Status and Headers

Next I'll cover how to change the status or the headers in a piece of middleware. The WSGI application calls start_response() to set the status and headers, so all the middleware has to do is provide its own start_response() function that performs any modifications before calling the start_response() function it has been passed. This example simply sets a cookie named name with the value value:

```
class Middleware(object):
    def __init__(self, app):
        self.app = app

    def __call__(self, environ, start_response):

        def custom_start_response(status, headers, exc_info=None):
            headers.append(('Set-Cookie', "name=value"))
            return start_response(status, headers, exc_info)

        return self.app(environ, custom_start_response)
```

Notice that custom_start_response() returns the value of calling the original start_response(). This is very important so that the writable returned from the application calling the function it receives as start_response() can use the writable object returned by the server. It is also important that you remember to pass the custom start response callable as the second argument when calling the WSGI application, rather than the original start_response() the middleware receives.

Handling Errors

To be able to deal with errors, you need to know that the start_response() callable you have been using throughout this chapter takes an optional third parameter named exc_info, which defaults to None.

If an error occurs in your application (or even in one of the middleware components), you might want to have that error intercepted so that you could log the error, send an e-mail error report, or display a traceback to the user for debugging. It is possible that the server hasn't actually sent the HTTP headers yet, so the middleware doing the intercepting has an opportunity to call start_response() itself with any status or headers it needs in order to display the traceback or indicate an error. Since the error-handling middleware doesn't know whether the server has actually sent the headers, it behaves as if it hasn't and calls the server as usual to generate the required response, but it also specifies the exc_info argument to start_response() as a Python sys.exc_info() tuple representing the error that occurred.

When start_response() is called with the exc_info argument, the server will check whether the headers have already been sent and raise the exception passed to it via the exc_info argument if they have. If they haven't, it can use the new headers specified by the error-handling middleware.

If you refer to the run_with_cgi() example from earlier in the chapter, you can see this behavior is implemented in the start_response() callable:

```
def start_response(status,response_headers,exc_info=None):
    if exc_info:
        try:
            if headers_sent:
                # Re-raise original exception if headers sent
                raise exc_info[0], exc_info[1], exc_info[2]
        finally:
            exc_info = None     # avoid dangling circular ref
    elif headers_set:
        raise AssertionError("Headers already set!")

    headers_set[:] = [status,response_headers]
    return write
```

In this example, if the headers have already been set and there is no `exc_info()` argument, an `AssertionError` is raised to indicate that it is likely the `start_response()` callable has been called twice by mistake.

With a good understanding of how the `exc_info` argument works, let's go ahead and write an example that simply displays a traceback using the `cgitb` module from the Python Standard Library:

```
import cgitb
import sys
from StringIO import StringIO

class Middleware(object):
    def __init__(self, app):
        self.app = app

    def format_exception(self, exc_info):
        dummy_file = StringIO()
        hook = cgitb.Hook(file=dummy_file)
        hook(*exc_info)
        return [dummy_file.getvalue()]

    def __call__(self, environ, start_response):
        try:
            app_iter = self.app(environ, start_response)
            for data in app_iter:
                yield data
        except:
            exc_info = sys.exc_info()
            start_response(
                '500 Internal Server Error',
                [('content-type', 'text/html')],
                exc_info
            )
            for data in self.format_exception(exc_info):
                yield data
        else:
            # Calling .close() could cause an exception too
            # so in a real handler you might test for that too
            if hasattr(app_iter, 'close'):
                app_iter.close()
```

As you can see, the middleware just uses a simple `try... except` block, and if it encounters an error, it calls `start_response()` with the `exc_info` argument so that the server can raise an exception if the headers have already been sent. Then it calls its `format_exception()` method, which uses the `cgitb` module to generate an HTML error page that it then returns.

Altering the Response

Another thing you can do with WSGI middleware is alter the response returned from a WSGI application. In Chapter 15, I discussed ways of speeding up the time it takes a browser to render a page from a Pylons application, and one of the recommendations was to compress JavaScript or CSS files using Gzip compression so that they took less time to download from a server. One way of doing this is with middleware, so let's write some middleware to Gzip JavaScript and CSS files before they are sent to the browser.

The example is quite complex, so rather than showing you the final code, let's build up the middleware in steps. First you need to know how to use Gzip compression in Python. Here's an example function that just compresses its input:

```
import gzip
import StringIO

def compress(string, compresslevel=9):
    # The GZipFile object expects to operate on a file, not a string
    # so we create a file-like buffer for it to write the output to
    buffer = StringIO.StringIO()

    # Now let's create the GzipFile object which compresses any
    # strings written to it and adds the output to the buffer
    output = gzip.GzipFile(
        mode='wb',
        compresslevel=compresslevel,
        fileobj=buffer
    )
    output.write(string)
    output.close()

    # Finally we get the compressed string out of the buffer
    buffer.seek(0)
    result = buffer.getvalue()
    buffer.close()

    return result
```

Let's make a first attempt at applying this technique to some middleware:

```
import gzip
import StringIO

# CAUTION: This doesn't work correctly yet

class GzipMiddleware(object):
    def __init__(self, app, compresslevel=9):
        self.app = app
        self.compresslevel = compresslevel

    def __call__(self, environ, start_response):
        buffer = StringIO.StringIO()
        output = gzip.GzipFile(
            mode='wb',
            compresslevel=self.compresslevel,
            fileobj=buffer
        )
        app_iter = self.app(environ, start_response)
        for line in app_iter:
            output.write(line)
        if hasattr(app_iter, 'close'):
            app_iter.close()
        output.close()
        buffer.seek(0)
        result = buffer.getvalue()
        buffer.close()
        return [result]
```

When the middleware is called, the `__call__()` method behaves in a similar way to the `compress()` function from the previous example. A buffer is set up to receive the compressed data, and a `GzipFile` object is created to do the work of compressing any data passed to it via its `write()` method. The WSGI application the middleware wraps is then called, and its result is iterated over, writing each line to the `GzipFile` object `output`. Notice that any time you iterate over the result, you are also responsible for calling the `close()` method on the iterator if it has one. This is to support resource release by the application and is intended to complement PEP 325's generator support and other common iterables with `close()` methods. Once the iterator has been closed, the method returns the compressed contents of the buffer to the middleware beneath it as a list.

The example so far correctly generates a compressed response, but there are some problems with it:

- Not all browsers support Gzip compression, so this middleware would break the application on those browsers.
- Any data written to the writable object returned by `start_response()` wouldn't be compressed.
- If a `Content-Length` was set, it would now be incorrect because the response has changed.
- Not all content should be compressed; only JavaScript and CSS files should be.

Let's start by writing a custom `start_response()` function that returns the `GzipFile` object. This object has a `write()` method which can be returned to fulfil the requirement of the WSGI specification for `start_response()` to return a writable object. At the same time, update the code to check that the browser supports Gzip compression. The new code looks like this with modified lines in bold:

```
import gzip
import StringIO

# CAUTION: This doesn't work correctly yet

class GzipMiddleware(object):
    def __init__(self, app, compresslevel=9):
        self.app = app
        self.compresslevel = compresslevel

    def __call__(self, environ, start_response):
        if 'gzip' not in environ.get('HTTP_ACCEPT_ENCODING', ''):
            return self.app(environ, start_response)
        buffer = StringIO.StringIO()
        output = gzip.GzipFile(
            mode='wb',
            compresslevel=self.compresslevel,
            fileobj=buffer
        )

        def dummy_start_response(status, headers, exc_info=None):
            return output.write

        app_iter = self.app(environ, dummy_start_response)
        for line in app_iter:
            output.write(line)
        if hasattr(app_iter, 'close'):
            app_iter.close()
        output.close()
        buffer.seek(0)
        result = buffer.getvalue()
        buffer.close()
        return [result]
```

If the browser doesn't support Gzip encoding, the middleware does nothing, returning the application as it stands. If Gzip encoding is supported, a `dummy_start_respose()` function is returned that, when called by the application, returns the object `output`. Although this code fixes two of the problems, it introduces a third. The `start_response()` callable that is passed to the `__call__()` method isn't called itself, so the headers and status won't actually get passed to the server, which you'll recall is responsible for providing `start_response()` in the first place. To call the `start_response()` function, you'll need to know the status and headers it should be called with.

The problem here is that `status`, `headers`, and `exc_info` are only locally available in your `dummy_start_response()` function, and you need to be able to access them in the scope of the `__call__()` method in order to call `start_response()` after the WSGI application is called. If you were to set them as globals, they become globally available, not just available in the scope of the `__call__()` method, so you definitely don't want to do that. The solution is to create a list variable in the scope of the `__call__()` method and then append the `status`, `headers`, and `exc_info` to it from the scope of the `dummy_start_response()` function; then after the application has been called, the variables will have been set, so you can iterate over the result based on the values in the list. Let's update the code to call the real `start_response()` callable:

```
import gzip
import StringIO

# CAUTION: This doesn't work correctly yet

class GzipMiddleware(object):
    def __init__(self, app, compresslevel=9):
        self.app = app
        self.compresslevel = compresslevel

    def __call__(self, environ, start_response):
        if 'gzip' not in environ.get('HTTP_ACCEPT_ENCODING', ''):
            return self.app(environ, start_response)
        buffer = StringIO.StringIO()
        output = gzip.GzipFile(
            mode='wb',
            compresslevel=self.compresslevel,
            fileobj=buffer
        )

        start_response_args = []
        def dummy_start_response(status, headers, exc_info=None):
            start_response_args.append(status)
            start_response_args.append(headers)
            start_response_args.append(exc_info)
            return output.write

        app_iter = self.app(environ, dummy_start_response)
        for line in app_iter:
            output.write(line)
        if hasattr(app_iter, 'close'):
            app_iter.close()
        output.close()
        buffer.seek(0)
        result = buffer.getvalue()
        start_response(**start_response_args)
        buffer.close()
        return [result]
```

This version correctly calls `start_response()`, but two problems are left to fix. First you need to update the `Content-Length` header to contain the correct length of the new compressed content, and then you need to ensure that only JavaScript and CSS files are compressed. Since this example will be used in the SimpleSite application, let's enable compression for any URL that ends in `.js` or `.css`. Here's the final version of the code:

```
import gzip
import StringIO

class GzipMiddleware(object):
    def __init__(self, app, compresslevel=9):
        self.app = app
        self.compresslevel = compresslevel

    def __call__(self, environ, start_response):
        if 'gzip' not in environ.get('HTTP_ACCEPT_ENCODING', ''):
            return self.app(environ, start_response)
        if environ['PATH_INFO'][-3:] != '.js' and environ['PATH_INFO'][-4:] != '.css':
            return self.app(environ, start_response)
        buffer = StringIO.StringIO()
        output = gzip.GzipFile(
            mode='wb',
            compresslevel=self.compresslevel,
            fileobj=buffer
        )
        start_response_args = []
        def dummy_start_response(status, headers, exc_info=None):
            start_response_args.append(status)
            start_response_args.append(headers)
            start_response_args.append(exc_info)
            return output.write

        app_iter = self.app(environ, dummy_start_response)
        for line in app_iter:
            output.write(line)
        if hasattr(app_iter, 'close'):
            app_iter.close()
        output.close()
        buffer.seek(0)
        result = buffer.getvalue()
        headers = []
        for name, value in start_response_args[1]:
            if name.lower() != 'content-length':
                 headers.append((name, value))
        headers.append(('Content-Length', str(len(result))))
        headers.append(('Content-Encoding', 'gzip'))
        start_response(start_response_args[0], headers, start_response_args[2])
        buffer.close()
        return [result]
```

As you can see, this version will work correctly, but it wasn't exactly a piece of cake to write. Although changing the response is one of the more difficult things you can do with middleware, it really makes you appreciate all the things Pylons does for you!

Testing the Gzip Middleware

Let's test this middleware. Add the previous code to a new file in the SimpleSite project's `lib` directory called `middleware.py`. Then edit the project's config file to include this import at the top:

```
from simplesite.lib.middleware import GzipMiddleware
```

Then change the code at the end of `config/middleware.py` so that it looks like this:

```
# Static files (If running in production, and Apache or another web
# server is handling this static content, remove the following 2 lines)
static_app = StaticURLParser(config['pylons.paths']['static_files'])
app = Cascade([static_app, app])
app = GzipMiddleware(app, compresslevel=5)
return app
```

If you start the SimpleSite server on your local machine and reload one of the pages that uses JavaScript such as `http://localhost:5000/page/edit/6`, you might find it actually loads more slowly than it did before. Take a look at the Net tab of Firebug to find out. This is because it is running on your local computer, so your machine has to do the extra work of compressing and uncompressing the data. If you were viewing the page over a network, the extra time it takes your browser to uncompress the data should be less than the time saved in sending the file across the network.

Here are the relevant headers from Firebug when the `yahoo-dom-event.js` file is requested before the Gzip middleware is in place:

```
Server            PasteWSGIServer/0.5 Python/2.5.2
Date              Fri, 10 Oct 2008 19:31:22 GMT
Content-Type      application/x-javascript
...
Content-Length    31637
```

Here are the headers with the Gzip middleware:

```
Server            PasteWSGIServer/0.5 Python/2.5.2
Date              Fri, 10 Oct 2008 19:30:44 GMT
Content-Type      application/x-javascript
...
Content-Length    10577
Content-Encoding  gzip
```

As you can see, the Gzip middleware has reduced the file size by about 20KB, which on a very slow connection could save up to half a second in the page load time.

Summary

This chapter has been a whistle-stop tour of the Web Server Gateway Interface. I hope you can see that WSGI is actually a very powerful API. If you managed to follow all the examples in the chapter, you might be keen to start writing WSGI middleware. If you are, that's great. A good place to start is the WSGI specification defined in PEP 333, but WSGI is best learned from examples. The Paste package that is installed with Pylons is particularly rich in WSGI middleware and is a good place to look to see how different situations are handled with WSGI.

Here are some links for further reading:

- PEP 333, `http://www.python.org/dev/peps/pep-0333/`
- The WSGI web site, `http://wsgi.org`
- Introducing WSGI—Python's Secret Web Weapon Part 1, `http://www.xml.com/pub/a/2006/09/27/introducing-wsgi-pythons-secret-web-weapon.html`
- Introducing WSGI—Python's Secret Web Weapon Part 2, `http://www.xml.com/pub/a/2006/10/04/introducing-wsgi-pythons-secret-web-weapon-part-two.html`