CHAPTER 20

Logging

A useful way of working out what is happening during a request is to write log statements at various points in your code. These log statements can output log messages to multiple places such as the console or a file to help you follow how your code is working.

Tip If you don't want to know the details of how logging works but are just desperate to log a message to the console from within a controller action, you can do so like this:

```
log.debug('Your message goes here')
```

Getting Started with Pylons Logging

Conceptually, it' helpful to think about Pylons applications as having two completely different types of logs:

- Server logs
- Application logs

Server logs are generated by the server running the Pylons application and will typically include information such as the URL that was requested and the time the request was made. The server might also log information such as the IP address of the user visiting the site.

The application logs are generated by your Pylons application and the packages it uses. Application log messages can come from your controllers, model, helpers, templates, or any other part of your application as well as from third-party packages your application uses such as SQLAlchemy or AuthKit. Application log messages are sometimes also sent to the server for logging, and this results in the application log messages being intermingled with the server log messages. As you can imagine, logging can quickly get quite complicated unless you keep the concept of a server log very separate from the concept of an application log in your own mind.

To demonstrate the difference between the two types of logs and how they interact in development mode with the Paste HTTP server, let's create a new project called `LogTest` with a controller named `log`. Choose Mako as the templating language, and you won't need SQLAlchemy or Google App Engine support:

```
$ paster create --template=pylons LogTest
$ cd LogTest
$ paster controller log
```

Now edit `logtest/controllers/log.py` so that the `index()` action looks like this:

```
def index(self):
    log.debug('My first Pylons log message!\n')
    return 'Check the logs!'
```

Start the server with the `--reload` and `--log-file` options like this:

```
$ paster serve --reload development.ini --log-file test.log
```

The `--log-file` option specifies the file that the log messages should be written to. Now visit `http://localhost:5000/log/index` in your browser. You should see the message `Check the logs!` in the browser, and you will see the message `My first Pylons log message!` has been written to `test.log`.

Now, with the server still running, add a new action to the `log` controller called `newlog()` that looks like this:

```
def newlog(self, action):
    log.debug("Logged from the 'newlog' action")
    return 'Check the logs again!'
```

The server will restart because you changed the file. Visit `http://localhost:5000/log/newlog`, and then stop the server. Once again, the message returned from the controller will be shown in the browser, and the log message will be written to `test.log`.

After having visited just these two URLs, the Paste HTTP server log file (`test.log`) will contain these lines:

```
serving on http://127.0.0.1:5000
12:49:46,347 DEBUG [logtest.controllers.log] My first Pylons log message!

/home/james/LogTest/logtest/controllers/log.pyc changed; reloading...
Starting server in PID 533.
serving on http://127.0.0.1:5000
12:50:29,567 DEBUG [logtest.controllers.log] Logged from the 'newlog' action
```

As you can see, the log file has two types of messages: the debug messages from the controller and the messages from the server. The server logs (in normal font) and the application logs (in bold) are combined into one file. So, let's discuss exactly what is happening in this example.

When the `log.debug()` function is called, a message is logged, but what happens to that message depends on the configuration you have set up with the logging options in the `development.ini` config file. The default settings cause all log messages of `INFO` level or above to be sent to the standard error stream.

Now, it just so happens that the Paste HTTP server captures all the information sent to the standard error stream to its server logs, so in this example both the server logs and the application logs end up combined in the same file. This is handy for development use, but not all servers will behave in the same way, so you can't guarantee application log output will always end up in the server logs with the default Pylons configuration, although it often will.

In the section "Logging to a File" later in the chapter, you'll learn how to redirect the application logs to a separate file, but before I go into too much detail, you need to learn a little bit more about the `logging` module used to produce the application logs.

Understanding the logging Module

Pylons uses Python's `logging` module to provide its application logging. The basic concept behind the `logging` module is that each message is logged to a particular logger and that each logger has a name. Each controller therefore has the following lines at the top to set up a logger to be used for logging messages related to each controller:

```
import logging
...
log = logging.getLogger(__name__)
```

Python's special `__name__` variable refers to the current module's fully qualified name, which in this case is `logtest.controllers.log`. This means that the `log` variable is set up to log messages to a logger named `logtest.controllers.log`.

The named loggers exist in a hierarchy where parent loggers can respond to the log messages of child loggers, and each . character in the name separates a level in the hierarchy. If, for example, you created a second logger in the controller specifically for logging the `index()` action and you named it `logtest.controller.log.index`, the `logtest.controller.log` logger would also receive its messages. This behavior is called *propagation* and can be overridden, as you'll see when it is discussed in more detail later in this chapter.

Tip The loggers don't need to have the names of the modules they are created in; they can have any names as long as you understand that they will be treated as part of a logger hierarchy with `.` characters separating parents from children. Most of the time, using the full name of the module in which the loggers are defined is a good idea, though.

In addition to ordinary loggers, there is also a *root logger* at the very top of the logging hierarchy. In the default Pylons setup, the individual loggers are not configured to handle their own log messages, so they get propagated to the root logger, which handles them instead. You'll learn more about this behavior later in this chapter.

Understanding Log Levels

Each message logged to a specific logger must also have a log level. Table 20-1 shows the main log levels and their corresponding numeric values.

Table 20-1. *Levels of Importance and Their Corresponding Numeric Values*

Level	Numeric Value
`CRITICAL`	50
`ERROR`	40
`WARNING`	30
`INFO`	20
`DEBUG`	10
`NOTSET`	0

Messages that you want to be displayed only during debugging might be assigned to the `DEBUG` level, whereas critical error messages should be logged to the `CRITICAL` level.

To log messages, you simply use one of the methods corresponding to the levels in Table 20-1, which are available on the `log` object you want to log messages for. Here's an example:

```
def index(self):
    log.error('My third Pylons log message!')
    return 'Check the logs!'
```

When this action is executed, the following will be logged:

```
16:20:20,440 ERROR [logtest.controllers.log] My third Pylons log message!
```

Notice that because you called `log.error()`, the message was logged with the `ERROR` level. Let's also try to log a simple warning message. Change the action you were editing earlier to look like this:

```
def index(self):
    log.warning('My third Pylons log message!')
    return 'Check the logs!'
```

If you restart the Paste HTTP server and visit the `http://localhost:5000/log/index` URL again, you will see that the message is displayed on the console with the `WARNING` level:

```
16:21:30,410 WARNI [logtest.controllers.log] My third Pylons log message!
```

It is also possible to log messages by their output number rather than by their level; you can specify any number from 0–50, not just numbers corresponding to a level. This is useful if you want to set your own fine-grained log levels. You can log a message at a particular numeric value like this:

```
def index(self):
    log.log(10, 'Another log message')
    return 'Check the logs!'
```

Logging Variables

Each of the logging methods `debug()`, `info()`, `warning()`, `error()`, `critical()`, and `log()` also accepts an optional set of variable names that will be substituted into the log message itself using standard Python string formatting. This means you can log variables like this:

```
def index(self):
    error = 'Wrong Number'
    value = 5
    log.error('The error %r occurred with a value of %s.', error, value)
    return 'Check the logs!'
```

When this action is executed, the following output will be logged:

```
16:20:20,440 ERROR [logtest.controllers.log] The error 'Wrong Number' occurred ➥
with a value of 5.
```

Logging in Templates

The `logging` module's `getLogger()` function always returns the same logger instance for the name it is given. This means that if you wanted to log to the `logtest.controllers.log` logger from within a template, you could access the same logger like this:

```
<%!
    import logging
    log = logging.getLogger('logtest.controllers.log')
%>
```

Then later in your code you could write a log message like this:

```
<% log.debug('This is a debug message') %>
```

Both the `log` object in the template and the `log` object in the controller log messages to the same log. Of course, you could also attach the controller's `log` object to `c.log` and access that directly in the template instead if you prefer.

Now that you've seen the basics of how the logging module works and you understand the difference between server logs and application logs in the context of a Pylons application, let's take a look at how logging is configured.

Introducing Logging Configuration

Pylons logging is configured through the Pylons config file and is used to change how the *application logs* are handled. The format used is the same as that used by the logging module, as described at http://docs.python.org/lib/logging-config-fileformat.html. Although it looks similar to the Paste Deploy configuration you learned about in Chapter 17, it is actually completely different.

Logging configuration consists of three types of section: *loggers*, *handlers*, and *formatters*. Broadly speaking, the formatters take a message and format it together with extra information available such as the time or the process ID of the running application. The handlers take the formatted messages and handle them in some way, perhaps by writing them to a file or sending them to the standard error stream, and the logger sections override the default setting of each of the loggers that are used to output log messages, either discarding them or passing them onto the appropriate handler depending on the severity level of the message. Each logger, handler, and formatter requires its own section in the config file, but to keep track of the names being used for each section, every config file must also contain sections called [loggers], [handlers], and [formatters] that identify the name and the type of each section in the file through the use of keys.

Here are the first sections defined in the LogTest project's development.ini file:

```
# Logging configuration
[loggers]
keys = root, routes, logtest

[handlers]
keys = console

[formatters]
keys = generic
```

This means the logging module will also expect to see sections named [logger_root], [logger_routes], [logger_logtest], [handler_console], and [formatter_generic]. The [logger_logtest] section is named according to the package name of your Pylons application, so the section name in your own project would reflect its package name instead of LogTest's.

■ **Tip** If you want to supplement the logging configuration supplied by the development.ini file, you can also configure logging in Python code. You might want to do this if you have written your own handler or formatter. A good place to add your extra configuration would be your project's environment.py file.

Let's take a look at the options you can use with each of the different types of section.

Logger Sections

Logger sections specify how to log messages to a particular logger, such as the logtest.controllers.log logger used earlier. Loggers take four configuration options. The level option specifies the log level below which log messages should be ignored. The handlers option takes a comma-separated list of the names of handlers to which the messages should be sent. The qualname option is the name of the logger to log messages for; and the propagate option determines whether messages sent to this logger should also be sent to its parent logger.

As an example, here's what the `routes` logger from the `LogTest` project's `development.ini` file looks like:

```
[logger_routes]
level = INFO
handlers =
qualname = routes.middleware
# "level = DEBUG" logs the route matched and routing variables.
```

This logger doesn't have any handlers of its own, but its messages propagate to the root logger, where they are handled instead.

Handler Sections

Handlers are used to handle the log messages passed to them from the loggers you have configured. Here's the console handler as an example:

```
[handler_console]
class = StreamHandler
args = (sys.stderr,)
level = NOTSET
formatter = generic
```

The `class` option indicates the handler's class (as determined by executing Python's `eval()` function in the `logging` package's namespace). The `args` argument is a list of arguments to pass to the handler specified by `class`. It's important to remember to add a comma at the end of the args list if there is just one argument; otherwise, the brackets are treated as parentheses rather than marking the start and end of a tuple.

You can use many handlers besides `StreamHandler` including `FileHandler`, `RotatingFileHandler`, `TimedRotatingFileHandler`, `SocketHandler`, `DatagramHandler`, `SysLogHandler`, `NTEventLogHandler`, `SMTPHandler`, `MemoryHandler`, and `HTTPHandler`. They are all documented in detail at `http://docs.python.org/lib/node409.html`.

The `level` option determines which level of messages are passed to the formatter and can be any one of the levels mentioned earlier: `CRITICAL`, `ERROR`, `WARNING`, `INFO`, `DEBUG`, and `NOTSET`. Setting the level to `NOTSET` results in everything being logged, setting the level to `INFO` results in messages of `INFO` level and above being logged. Finally, the `formatter` option should be the name of a formatter section that will specify how the log message should be formatted.

Formatter Sections

Formatters are responsible for converting a log record passed from a handler to a format suitable for output, usually a string with certain extra data about when and where the message was logged.

Here is an example:

```
[formatter_form01]
format=F1 %(asctime)s %(levelname)s %(message)s
datefmt=
class=logging.Formatter
```

The `format` option is the overall format string, and the `datefmt` option is the `strftime()`-compatible date/time format string. If it is empty, the `logging` package substitutes ISO8601 format dates and times.

The `class` entry is optional. It indicates the name of the formatter's class (as a dotted module and class name) and is useful if you have created your own `Formatter` subclass you want to use.

Most of the time, the only line in formatter sections that you'll want to change is the `format` line. You can use the variables in Table 20-2 to configure how the log messages will be formatted.

Table 20-2. *Format Codes and Their Descriptions*

Format	Description
`%(name)s`	Name of the logger.
`%(levelno)s`	Numeric logging level for the message (`DEBUG`, `INFO`, `WARNING`, `ERROR`, `CRITICAL`).
`%(levelname)s`	Text logging level for the message (`'DEBUG'`, `'INFO'`, `'WARNING'`, `'ERROR'`, `'CRITICAL'`).
`%(pathname)s`	Full path name of the source file where the logging call was issued (if available).
`%(filename)s`	File name portion of path name.
`%(module)s`	Module (name portion of file name).
`%(funcName)s`	Name of function containing the logging call. Added in Python 2.5.
`%(lineno)d`	Source line number where the logging call was issued (if available).
`%(created)f`	Time when the `LogRecord` was created (as returned by `time.time()`).
`%(relativeCreated)d`	Time in milliseconds when the `LogRecord` was created, relative to the time the logging module was loaded.
`%(asctime)s`	Human-readable time when the `LogRecord` was created. By default, this is of the form 2003-07-08 16:49:45,896 (the numbers after the comma are millisecond portion of the time).
`%(msecs)d`	Millisecond portion of the time when the `LogRecord` was created.
`%(thread)d`	Thread ID (if available).
`%(threadName)s`	Thread name (if available).
`%(process)d`	Process ID (if available).
`%(message)s`	The logged message, computed as `msg % args`.

The formatter section used by `LogTest` looks like this:

```
[formatter_generic]
format = %(asctime)s,%(msecs)03d %(levelname)-5.5s [%(name)s] %(message)s
datefmt = %H:%M:%S
```

You can work out what each of the variables in the `format` option does from Table 20-2.

Over the next sections, you'll learn how to tweak the settings in each of the types of sections to solve common logging problems, so don't worry if you don't understand all the options just yet.

Tip If you want to understand logging in detail, it is well worth reading the `logging` documentation at `http://docs.python.org/lib/module-logging.html`.

Redirecting Log Output Using Handlers

Now that you have seen the options for each of the different types of logging configuration sections, let's look at some of the common ways of handling log messages.

One of the most useful things to do with application log messages is to write them directly to a file. In fact, if you are new to logging or are setting up a Pylons applications on a new server, it is highly recommended you start by logging messages to a file because there is a lot less that can go wrong compared with logging either to the standard output stream, the standard error stream, or the WSGI errors stream.

Logging to a File

Although it is helpful to see application log messages in the console when you are developing a Pylons application, for production use the messages often need to be logged to a file.

To capture log output to a separate file, you can use a `FileHandler` or a `RotatingFileHandler`. The following is the configuration to set up a `FileHandler` called `file`. Add it to the end of the `development.ini` file from the `LogTest` project you created at the start of the chapter:

```
[handler_file]
class = FileHandler
args = ('application.log', 'a')
level = INFO
formatter = generic
```

The options here are similar to those used for the `console` handler that already exists in the `development.ini` file. The `args` option specifies the file name of the log file and that it should be opened in *append mode* so that new information is added to the end of the file and doesn't overwrite existing data. The new handler will use the same formatter as the console handler.

For Pylons to know this section represents a new log handler, you also have to add the `file` handler to the `[handlers]` section. Update it to look like this:

```
[handlers]
keys = console, file
```

Now that the new handler is set up, you can customize the `root` logger to use the `file` handler rather than the existing `console` handler. Change the `[logger_root]` section to look like this:

```
[logger_root]
level = INFO
handlers = file
```

Now all the root logger's messages will be directed to the `application.log` file instead of the `sys.stderr` error stream.

With the new handler in place, start the Paste HTTP server again, this time using the file name `server.log` for the output log:

```
$ paster serve --reload development.ini --log-file=server.log
```

If you visit `http://localhost:5000/log/index`, you will see that the application log is not sent to `server.log` but instead appears in `application.log`. The server messages continue to appear in `server.log`, though. You have successfully redirected the application logs to a file.

■ **Tip** If you are using log files, it can often be useful to see the output that is logged as it is written, without having to constantly close and reopen the file. On Unix platforms you can use this command:

```
$ tail -f server.log
```

With this command running, any messages will be automatically copied to the console as they are written to `server.log` so that you can see the messages appear as you interact with the server.

Logging to wsgi.errors

You'll remember from Chapter 16 that one of the responsibilities of a WSGI server is to provide a suitable writable object as `environ['wsgi.errors']` to be used for logging errors. Because of this, any messages written to `wsgi.errors` are guaranteed to go to the server's error log no matter which server you are using and no matter what format that error log takes (as long as the server conforms to the WSGI specification, of course).

You can test the behavior of `wsgi.errors` with the Paste HTTP server by adding a new action to the log controller:

```
def wsgi_errors(self):
    request.environ['wsgi.errors'].write(
        'This is sent directly to the wsgi.errors stream')
    return 'Message logged to wsgi.errors'
```

Check that the server is still running and has the `--log-file server.log` option set. Now visit `http://localhost:5000/log/wsgi_errors`, and you will see `Message logged to wsgi.errors` returned to the browser. If you look at `server.log`, you will see the last line looks like this:

```
This is sent directly to the wsgi.errors stream
```

As you can see, sending messages to the `wsgi.errors` stream sends messages directly to the *server* log, not the *application* log and so bypasses all the standard logging configuration. As a result, the extra information added by a formatter such as the time or log level has not been added.

Now that you know how the `wsgi.errors` stream logs messages, you might decide you want to be able to use it via a handler in the configuration file using the existing logging infrastructure. Pylons provides a custom logging handler class specifically for this purpose called `WSGIErrorsHandler`, which you can use to automatically send log messages to the `wsgi.errors` stream. It is documented at `http://docs.pylonshq.com/modules/log.html`. The advantage of using `WSGIErrorsHandler` is that all application logs get mixed in with the server logs, so you have only one log file to deal with, and you can quickly see the order that particular server and application events occurred in without having to compare their time stamps.

■ **Caution** Although using the `wsgi.errors` stream can be useful, there is one big problem with using the Pylons `WSGIErrorsHandler`: all messages logged outside of a Pylons request will be silently lost.

The `wsgi.errors` stream is available only during a request because it has to be accessed via the WSGI `environ` dictionary, which itself is available only during a request. This means log messages created during application startup, shutdown, or before and after a request are silently lost.

This silent loss of messages may be something you can tolerate in your application, but generally speaking it is better to handle the application logs via a `FileHandler` (described in the previous section) or a `RotatingFileHandler`, so you can be sure all the messages are logged.

If you still want to use the `WSGIErrorsHandler`, this is how you do it. First add the handler name `wsgierrors` to the `[handlers]` section of the configuration file so that Pylons knows it is a handler:

```
[handlers]
keys = console, file, wsgierrors
```

Then add a new handler section specifying `pylons.log.WSGIErrorsHandler` as the class and an empty tuple for `args`. In this example, only messages with the `DEBUG` level or higher are logged, and you are using the same `generic` formatter you used for the `console` and `file` handlers.

```
[handler_wsgierrors]
class = pylons.log.WSGIErrorsHandler
args = ()
level = DEBUG
format = generic
```

You could now update the root logger's handler to use the `wsgierrors` handler on its own, but it is safer to use both the `file` handler and the `wsgierrors` handlers together so that any messages ignored by the `wsgierrors` handler are at least captured by the `file` handler. Change the `[logger_root]` section to look like this:

```
[logger_root]
level = INFO
handlers = file, wsgierrors
```

If you try this, notice that application messages are logged both to the Paste HTTP server log (via `wsgierrors`) as well as directly to `application.log`. If you log any messages outside a request, they will appear only in the `application.log` file via the `file` handler.

Configuring Which Messages Are Logged

Now that you've seen a couple of examples of how to use different handlers, it's time to see how you can use handlers to control which messages are logged. The golden rule is that the level that eventually gets logged is the higher of the level specified in the logger and the level specified in the handler.

This means that if you want only `ERROR`-level messages and above going to the `wsgi.errors` stream, you can change its handler definition to look like this:

```
  [handler_wsgierrors]
  class = pylons.log.WSGIErrorsHandler
  args = ()
  level = ERROR
  format = generic
```

Now only the messages that are logged at the `ERROR` level or above get sent to the `wsgi.errors` stream by the `pylons.log.WSGIErrorsHandler` handler.

Controlling Propagation with Loggers

Now that you've seen some of the things that you can do with handlers, let's look in more detail at loggers.

By default, three loggers are configured for your `development.ini` configuration. In the case of `LogTest`, the relevant lines look like this:

```
[loggers]
keys = root, routes, logtest

...

[logger_root]
level = INFO
handlers =

[logger_routes]
level = INFO
handlers =
qualname = routes.middleware
# "level = DEBUG" logs the route matched and routing variables.
```

```
[logger_logtest]
level = DEBUG
handlers =
qualname = logtest

...
```

As you can see, loggers are configured for the following:

- All messages (`logger_root`)
- Routes middleware messages (`logger_routes`)
- Messages from the `LogTest` project (`logger_logtest`)

If you look at the `[logger_logtest]` section, you'll see that no handler is defined, so you might be wondering how the test messages you wrote in the `log` controller at the beginning of the chapter were logged to the application log. The answer is via *propagation*.

If you don't explicitly set a value for the `propagate` option, it is assumed to be set to `1`. This means that any messages sent to the logger are *also* propagated to its parent logger. In this case, the parent logger is the root logger, and this does have a handler specified, so all messages to the `logtest` logger are actually handled by the root logger.

■ **Caution** You have to be very careful when spelling `propagate` because Pylons won't give you a warning if you misspell it. I've spent quite a long time wondering why propagation wasn't working only to realize I'd made a typo, and I wouldn't want you to make the same mistake!

You can test this behavior by setting `propagate` to `0` in the configuration section for `[logger_logtest]` and restarting the server. If you do this and then visit the `http://localhost:5000/log/index` URL, you'll see the following error message instead of the log message you might have expected:

```
No handlers could be found for logger "logtest.controllers.log"
```

This is because the messages are no longer propagated to the root logger and no handlers are specified for the `logtest.controllers.log` logger. The logging system is warning you that you might have made a mistake.

You might be wondering how the messages were sent to the `logtest` logger in the first place when the logger the messages were sent to is actually called `logtest.controller.log`. Once again, this occurs via propagation. Because the `qualname` option is specified with the value `logtest`, the `[logger_logtest]` configuration will apply to any logger whose name starts with `logtest.`. To test this, you could create a new logger in the log controller, which looks like this:

```
other_log = logging.getLogger('logtest.controllers.log.other')
```

Then in the `index()` action you could choose to use the new logger like this:

```
def index(self):
    other_log.info("Logged with the 'logtest.controllers.log.other' logger")
    log.info("Logged with the %r log", __name__)
    return 'Check the logs!'
```

If you tested the example, you'd see that messages to both loggers are output:

```
21:30:43,621 INFO  [logtest.controllers.log.other] Logged with the ➥
'logtest.controllers.log.other' logger
21:30:43,622 INFO  [logtest.controllers.log] Logged with the ➥
'logtest.controllers.log' log
```

Since both the loggers in this example are children of the `logtest` logger, both get logged. Remember to change the `propagate` option back to `1` if you test this example so that the messages are propagated onto the root logger and then onto the handler.

■ Tip Remember that the logger name used in controllers is related to the `qualname` specified in the configuration for the logger section, not to the name of the config file section. This means you could rename the `[logger_logtest]` section to something completely different such as `[logger_foo]` as long as you kept the `qualname` as `logtest` and updated other references within the config file to use the `foo` name.

Using Propagation to Filter Messages

You can also use propagation to filter log messages. For example, say you wanted only `WARNING` messages or above from the `logtest.controllers.log.other` logger but still wanted `DEBUG` messages or above for all other children of the `logtest` logger. You could update the config file like this (changes are in bold):

```
[loggers]
keys = root, routes, logtest, logtest_controllers_log_other

...

[logger_logtest]
level = DEBUG
handlers =
qualname = logtest
propagate = 1

[logger_logtest_controllers_log_other]
level = WARNING
handlers =
qualname = logtest.controllers.log.other
propagate = 1
```

Now if you tested the previous example again so that an `INFO` message is logged to both the `logtest.controllers.log.other` logger and the `logtest.controllers.log` logger, you'd just see the message that was logged to `logtest.controllers.log`. This is because `logtest.controllers.log.other` is only accepting messages of `WARNING` level or above, so the `INFO` message it receives is ignored. Here is the output logged:

```
21:30:53,422 INFO  [logtest.controllers.log] Logged with the ➥
'logtest.controllers.log' log
```

If you updated the action to use a warning or error message like this, the log message would appear again, this time as a `WARNING`.

```
def index(self):
    other_log.warning("Logged with the 'logtest.controllers.log.other' logger")
    log.info("Logged with the %r log", __name__)
    return 'Check the logs!'
```

If you tested the example, you'd see the following output:

```
21:31:48,531 WARN  [logtest.controllers.log.other] Logged with the ➥
'logtest.controllers.log.other' logger
21:31:48,532 INFO  [logtest.controllers.log] Logged with the ➥
'logtest.controllers.log' log
```

A logger's `level` option does not affect messages which it receives via propagation from a child logger. This means the `level` option can only be used to filter messages which are received directly. To test this, update the config file so that the level for `logtest.controllers.log.other` is set to DEBUG and the level for `logtest.controllers.log` is set to `WARNING`. If you call `other_log.info ('This message will be logged')` in the controller you will see the message is logged because it isn't filtered by the `level = WARNING` option in the `[log_logtest]` section.

This is why the very first example in the chapter worked even though it logged a debug message and the root logger level was specified as `INFO`.

Summarizing Propagation Options

In practical terms, you can use propagation in three main ways:

- Set up nonroot loggers that have no handlers but do propagate. The non-root loggers can then be used to adjust the verbosity of their logging output by changing their level.
- You could also set up nonroot loggers that have handlers but that *do not* propagate. These produce output only in the handlers they specify and not in the root handler. If a handler has a level higher than the underlying message, the output is suppressed. You'll see an example of this setup next when you add SQLAlchemy output to a new log file.
- Some nonroot loggers have handlers and do propagate. The message will appear in both places. This can be useful if you are using the `WSGIErrorsHandler` to ensure that all messages get logged.

Capturing Log Output from Other Software

When you are trying to debug a problem in a particular controller, or perhaps a particular module from, say, SQLAlchemy, then you might find that setting the root logger to `DEBUG` or `NOTSET` will produce too many messages for you to easily deal with, most of which will be generated by modules you aren't interested in. Instead, you want to be able to adjust only the verbosity of output from one logger or set of loggers.

To do this, you need to add logging configuration for the logger you want to add log messages for. You can then choose to have them handled by the root logger using propagation, configure them to use a separate handler, or do both. Let's look at these approaches.

Capturing SQLAlchemy Log Messages Using Propagation

As an example, let's set up a logger to log a SQLAlchemy engine. To do this, you need to set up a new logger in the configuration file; name it `sqlalchemy`:

```
[loggers]
keys = root, routes, logtest, sqlalchemy
```

Now you need to add the configuration for that logger:

```
[logger_sqlalchemy]
level = DEBUG
handlers =
qualname = sqlalchemy.engine
propagate = 1
```

The qualname option here specifies that this logger will handle any messages sent to a logger named sqlalchemy.engine or any children it might have. Because the propagate option is set to 1, this logger's messages get propagated up to the root logger where they are handled by the root logger's handler.

Even if you set the root logger's level to something higher such as ERROR, the sqlalchemy.engine debug messages logged at the INFO and DEBUG levels would still be logged because, as you've seen, the level option on the root logger doesn't apply to log messages propagated from configured loggers.

If you chose to enable SQLAlchemy support when you used paster create to create your Pylons project, you'd see that your config file already has the following configuration set up:

```
[logger_sqlalchemy]
level = INFO
handlers =
qualname = sqlalchemy.engine
# "level = INFO" logs SQL queries.
# "level = DEBUG" logs SQL queries and results.
# "level = WARNING" logs neither.  (Recommended for production systems.)
```

By changing the qualname option, you can adjust which parts of SQLAlchemy are logged. By changing the log level of the sqlalchemy logger, you can adjust which SQLAlchemy messages are propagated to the root logger to be handled. The INFO level results in SQL queries being logged, and the DEBUG level causes both queries and results to be logged. You may decide that you want to turn off SQLAlchemy log messages temporarily. If so, you can leave the root logger set to INFO and set the sqlalchemy logger's level to WARNING, and then the usual INFO log messages will be suppressed.

Capturing AuthKit Messages Using a Handler

Another piece of software that uses log messages is AuthKit. Some of what AuthKit does behind the scenes is rather complicated, so if you are trying to debug a particular behavior, logging can help.

Let's configure AuthKit so that its messages are sent straight to application.log via the file handler. First add the logger to the [loggers] section and change the [logger_root] section so that it is no longer using the file handler itself:

```
[loggers]
keys = root, routes, logtest, authkit
[logger_root]
level = INFO
handlers = wsgierrors
```

Then add the configuration for the new logger:

```
[logger_authkit]
level = DEBUG
handlers = file
qualname = authkit
propagate = 0
```

In this case, the qualname is authkit to capture all AuthKit log messages. The handler is set to file to use the same file handler section you set up earlier, and propagate is set to 0 so that the log messages aren't propagated to the root logger.

In this configuration, the AuthKit messages get sent directly to the file handler where they are logged to the application.log file. Of course, nothing is stopping you from using a handler and propagating a message to the root logger. To do this, just set propagate to 1 and you'll see the message is passed to the root logger which passes it to the wsgierrors handler so that it gets logged there too.

Production Configuration

So far in this chapter I've been discussing the logging configuration for a development setup, but Pylons uses a different default configuration for production setups.

If you install the LogTest project you can create a production configuration file like this:

```
$ paster make-config LogTest production.ini
Distribution already installed:
  LogTest 0.1dev from /home/james/Desktop/LogTest
Creating production.ini
Now you should edit the config files
  production.ini
```

The logging part of the generated production.ini file looks like this:

```
# Logging configuration
[loggers]
keys = root

[handlers]
keys = console

[formatters]
keys = generic

[logger_root]
level = INFO
handlers = console

[handler_console]
class = StreamHandler
args = (sys.stderr,)
level = NOTSET
formatter = generic

[formatter_generic]
format = %(asctime)s %(levelname)-5.5s [%(name)s] %(message)s
```

As you can see, this defines just one logger, the root logger. The console handler is the same for the development setup, and the formatter is only slightly different (it doesn't include the millisecond part of the time).

This means that any messages of INFO level or above will be logged to the standard error stream. As you've seen, this is likely to send log messages generated by the application to the server log, so you might prefer to set up a file handler instead. You saw how to do this in the "Logging to a File" section.

Summary

In this chapter, you saw how to use logging within a Pylons controller and template and took a brief tour of some of the features of Python's logging module. You also saw how logging configuration is divided into sections for loggers, handlers, and formatters, and you saw the options each of these different sections can take and what the options do.

You also saw how to control log messages, filtering them by changing the log levels in a handler or using propagation to control which levels of messages get propagated. You saw how to direct certain messages to different handlers such as an external file, and you know how to configure logging to have messages from other software included in the logs.

With the knowledge you've gained, you should be able to take control of the Pylons log messages and make them work for you. When used correctly, logging can be a very powerful tool.