# 20. Serving HTTP - Python in a Nutshell, 3rd Edition

## Some Popular Lightweight Frameworks

As mentioned, Python has multiple frameworks, including many lightweight ones. We cover four of the latter: Falcon, Flask, Bottle, and webapp2. In our coverage of each framework, we provide an example toy app that greets the users and reminds them of when they last visited the site, based on a `lastvisit` cookie. First, we show how to set the cookie in the main "greeter" function. However, an app would presumably want to set the `lastvisit` cookie no matter how the app was getting "visited." Instead of repeating the cookie setting in each and every appropriate method, these frameworks let you easily factor out the code to respect the key design principle of DRY ("Don't Repeat Yourself"). So, we also provide an example of how to factor out the code for this purpose in each framework.

### Falcon

Possibly the fastest (thus, by inference, lightest) Python web framework is Falcon. According to the benchmarks on its site, it runs 5 to 8 times faster than the most popular lightweight Python web framework, Flask, on both v2 and v3. Falcon is very portable, supporting Jython, or Cython for a further little speed-up, or PyPy for a huge boost—the latter making it, in all, 27 times faster than `Flask`. In addition to the project website, look at the sources on GitHub, the installation notes on PyPI, and the extensive docs. To run `Falcon` in Google App Engine locally on your computer (or on Google's servers at *appspot.com*), see Rafael Barrelo's useful GitHub page.

The main class supplied by the `falcon` package is named `API`. An instance of `falcon.API` is a WSGI application, so a common idiom is a multiple assignment:

```
import falcon

api = application = falcon.API()
```

When you instantiate `falcon.API`, pass zero or more named parameters:

`media_type`

Content type used in responses unless otherwise specified; defaults to `'application/json; charset=utf-8'`

`middleware`

A *Falcon middleware* object, or a list of such objects, covered in "Falcon middleware"

`request_type`

The class to use for requests; defaults to `falcon.request.Request`, but you might code a subclass of that class and use the subclass here

`response_type`

The class to use for responses; defaults to `falcon.response.Response`, but you might code a subclass of that class and use the subclass here

`router`

The class to use for routing; defaults to `falcon.routing.CompiledRouter`, not suitable to inherit from, but you could code your own router class, make it compatible by duck typing, and use your router class here

Once you have the `api` object, you can customize it in several ways, to handle errors and to add *sinks (*using regular expressions to intercept and dispatch a set of routes, for example by proxying to another site). We do not cover these advanced topics (nor custom request, response, and router types) in this book: the Falcon docs (particularly, as a reference, the Classes and Functions section) cover them well.

One customization method that you almost invariably do call on `api`, usually more than once, is necessary to add routing, and thus to get requests served:

**add_route** `add_route(`*`uri_template`*`,`*`resource`*`)`

`uri_template` is a string matching the URI to route; for example, `'/'` to route requests that ask for the root URL of the site. The template may include one or more *field expressions*—identifiers enclosed in braces (`{}`)—in which case the *responder methods* of the corresponding `resource` must accept arguments named like those identifiers, in order, after the mandatory arguments for request and response.

`resource` is an instance of a *Falcon resource* class, as covered in "Falcon resources". Note that you pass a specific instance, which gets reused to serve all requests to the given URI, not the class itself: be very careful in your coding, especially about using instance variables.

## Falcon resources

A *Falcon resource class* is one that supplies methods, known as *responder methods*, whose names start with `on_` followed by an HTTP verb in lowercase; for example, the method `on_get` responds to the most common HTTP verb, GET.

Each responder method normally has a signature such as:

```
def on_get(self, req, resp): ...
```

accepting the request object and the response object as arguments. However, when the URI template routing to the resource includes field expressions, then responder methods need to accept extra arguments with names matching the identifiers in the field expressions. For example, if you route to the resource by calling `api.add_route('/foo/{bar}/{baz}', `*`rsrc`*`)`, then responder methods of the resource-class instance `rsrc` must have signatures such as:

```
def on_post(self, req, resp, bar, baz): ...
```

A resource does not have to supply responder methods for all HTTP verbs—if the resource is addressed by an HTTP verb it does not support (e.g., a POST to a route whose resource's class has no `on_post` method), Falcon detects that, and responds to the invalid request with HTTP status `'405 Method Not Allowed'`.

Falcon does not supply any class for resource classes to inherit from: rather, resource classes rely entirely on *duck typing* (i.e., they just define responder methods with the appropriate names and signatures) to fully support Falcon.

## Falcon request objects

The `falcon.Request` class supplies a large number of properties and methods, thoroughly documented in Falcon's docs. The ones you'll be using most often are property cookies, a `dict` mapping cookie names to values, and the methods `get_header` to get a header from the request (`None`, when the request had no such header), `get_param` to get a parameter from the request (`None`, when the request had no such parameter), and small variants such as `get_param_as_int` to get an `int` value for a request parameter that must be an integer.

### Falcon response objects

The `falcon.Response` class supplies many properties and methods, thoroughly documented in Falcon's docs. Usually, you'll need to set many of the properties (except for ones controlling headers you're okay with being absent or taking on default values), and, often, call many of the methods. Here are the properties:

`body`

Assign a Unicode string to encode in UTF-8 as the response body. If you have a bytestring for the body, assign it to `data` instead, for speed.

`cache_control`

Assign to set the Cache-Control header; when you assign a list of strings, Falcon joins them with `', '` to produce the header value.

`content_location`

Assign to set the Content-Location header.

`content_range`

Assign to set the Content-Range header.

`content_type`

Assign to set the Content-Type header; a typical value you'd assign to `resp.content_type` might be, for example, `'text/html; charset=ISO-8859-15'` when you're setting `resp.data` to an HTML document encoded in `ISO-8859-15` (defaults to `api.media_type`).

`data`

Assign a bytestring to use the bytes as the response body.

`etag`

Assign to set the ETag header.

`last_modified`

Assign to set the Last-Modified header.

`location`

Assign to set the Location header.

`retry_after`

Assign to set the Retry-After header, typically in conjunction with setting `resp.status` to `'503 Service Unavailable'`.

`status`

Assign a status-line string, defaulting to `'200 OK'` (for example, set `resp.status = '201 Created'` if an `on_post` method creates a new entity—see RFC 2616 for all details on HTTP status codes and status lines).

`stream`

Assign either a file-like object whose `read` method returns a string of bytes, or an iterator yielding bytestrings, when that's a more convenient way for you to express the response body than assigning to `resp.body` or `resp.data`.

`stream_len`

Optionally, assign an `int`, the number of bytes in `resp.stream`, when you assign the latter; Falcon uses it to set the Content-Length header (if not supplied, Falcon may use chunked encoding or other alternatives).

`vary`

Assign to set the Vary header.

## Lightweight frameworks: you must know what you're doing!

You may have noticed that properly setting many of these properties requires you to understand HTTP (in other words, to know what you're doing), while a full-stack framework tries to lead you by the hand and have you do the right thing without really needing to understand how or why it is right—at the cost of time and resources, and of accepting the full-stack framework's conceptual map and mindset. ("You pays yer money and you takes yer choice"!) The authors of this book are enthusiasts of the knowledge-heavy, resources-light approach of lightweight frameworks, but we acknowledge that there is no royal road to web apps, and that others prefer rich, heavy, all-embracing full-stack frameworks. To each their own!

Back to `falcon.Response` instances, here are the main methods of an instance `r` of that class:

Table 20-1.

| | |
|---|---|
| **add_link** | `r.add_link(target,rel,title=None,title_star=None, anchor=None,hreflang=None,type_hint=None)`<br><br>Add a Link header to the response. Mandatory arguments are `target`, the target IRI for the link (Falcon converts it to a URI if needed), and `rel`, a string naming one of the many relations catalogued by IANA, such as `'terms-of-service'` or `'predecessor-version'`. The other arguments (pass each of them as a named argument, if at all) are rarely needed, and are documented in the Falcon docs and RFC 5988.<br><br>If you call `r.add_link` repeatedly, Falcon builds one `Link` header, with the various links you specify separated by commas. |

| | |
|---|---|
| **append_header** | `r.append_header(name, value)` <br><br> Set or append an HTTP header with the given `name` and `value`. If a header of such `name` is already present, Falcon appends the `value` to the existing one, separated by a comma; this works for most headers, but not for *cookies*—for those, use the `set_cookie` method instead. |
| **get_header** | `r.get_header(name)` <br><br> Return the string value for the HTTP header with the given `name`, or `None` if no such header is present. |
| **set_cookie** | `r.set_cookie(name,value,expires=None,max_age=None,` `domain=None,path=None,secure=True,http_only=True)` <br><br> Set a `Set-Cookie` header on the response. Mandatory arguments are `name` (the cookie name) and `value`, a `str` (the cookie's value). The other arguments (pass each of them as a named argument, if at all) may be needed for extra security (and cookie persistence: if you set neither `expires` nor `max_age`, the cookie expires as soon as the browser session finishes), and are documented in the Falcon docs and RFC 6525. <br><br> When you call `r.set_cookie` more than once, Falcon adds multiple `Set-Cookie` headers to the response. |
| **set_header** | `r.set_header(name, value)` <br><br> Set an HTTP header with the given `name` and `value`. When a header with that `name` was already present, it's replaced. |
| **set_headers** | `r.set_headers(headers)` <br><br> Set multiple HTTP headers as per `set_header`. `headers` is either an iterable of `(name, value)` pairs, or a mapping with `name` as the key, `value` as the corresponding value. |
| **unset_cookie** | `r.unset_cookie(name)` <br><br> Remove the cookie named `name` from the response, if it's there; also add to the response a `Set-Cookie` header asking the browser to remove the cookie from its cache, if the cookie is in that cache. |

## Falcon middleware

A *Falcon middleware class* is one that supplies one or more of three methods that examine, and potentially alter, incoming requests and outgoing responses. You pass in a list of instances of such classes as the `middleware` argument when you instantiate `falcon.API`, and those instances are given a chance to examine and alter all the incoming requests and outgoing responses. Multiple middleware class instances in the list get that chance in first-to-last order on incoming requests, then backward, last-to-first, on outgoing responses.

Falcon does not supply any class for middleware classes to inherit from: rather, middleware classes rely entirely on *duck typing* (i.e., they just define one or more of the three methods with appropriate name and signature). If a middleware class does not supply one or more of the three methods, Falcon treats its instances as if they supplied an empty implementation of the missing method(s), doing nothing.

The three methods Falcon middleware classes may supply are:

| | |
|---|---|
| **process_request** | `process_request(self,`*`req`*`,`*`resp`*`)`<br><br>Called *before* routing; if it changes `req`, that may affect routing. |
| **process_resource** | `process_resource(self,`*`req`*`,`*`resp`*`,`*`resource`*`)`<br><br>Called *after* routing, but before processing. `resource` may be `None` if routing could not find a resource matching the request. |
| **process_response** | `process_response(self,`*`req`*`,`*`resp`*`,`*`resource`*`)`<br><br>Called after the responder method, if any, returns. `resource` may be `None` if routing could not find a resource matching the request. |

Example 20-1 shows a simple Falcon example.

### Example 20-1. A Falcon example

```
import datetime, falcon
one_year = datetime.timedelta(days=365)

class Greeter(object):
    def on_get(self, req, resp):
        lastvisit = req.cookies.get('lastvisit')
        now = datetime.datetime.now()
        newvisit = now.ctime()

        thisvisit = '<p>This visit on {}UTC</p>' .format(newvisit
)
        resp.set_cookie('lastvisit', newvisit,
                         expires=now+one_year, secure=False)
        resp_body = [
            '<html><head><title>Hello, visitor!
            </title>'
            '</head><body>']
        if lastvisit is None:
            resp_body.extend((
            '<p>Welcome to this site on your first visit!
            </p>'                                            ,
            thisvisit,
            '<p>Please Refresh the web page to
            proceed</p>'                                  ))
        else:
            resp_body.extend((
            '<p>Welcome back to this site!
            </p>'                             ,

            '<p>You last visited on {}UTC</p>' .format(lastvisit
),
            thisvisit))
        resp_body.append('</body></html>')
        resp.content_type = 'text/html'
        resp.stream = resp_body
        resp.stream_len = sum(len(x) for x in resp_body)

app = falcon.API()
greet = Greeter()
app.add_route('/', greet)
```

This Falcon example just shows how to use some of the fundamental building blocks that Falcon offers—the API, one resource class, and one instance, preparing the response (in this case, setting `resp.stream` is the natural way, since we have prepared in `resp_body` a list of string pieces of the response body; `resp.body = ''.join(resp_body)` would also be okay). The example also shows how to maintain a minimal continuity of state among multiple interactions with the server from the same browser by setting and using a cookie.

If this app had multiple resources and responder methods, presumably it would want to set the `lastvisit` cookie no matter through which resource and method the app was getting "visited." Here's how to code an appropriate middleware class and instantiate `falcon.API` with an instance of said class:

```
class LastVisitSettingMiddleware(object):
    def process_request(self, req, resp):
        now = datetime.datetime.now()
        resp.set_cookie('lastvisit', now.ctime(),
                            expires=now+one_year, secure=
False)

lastvisitsetter = LastVisitSettingMiddleware()

app = falcon.API(middleware=[lastvisitsetter])
```

You can now remove the cookie setting from `Greeter.on_get`, and the app will keep working fine.

## Flask

The most popular Python lightweight framework is Flask. Although lightweight, it includes a development server and debugger, and explicitly relies on other well-chosen packages such as Werkzeug and jinja2 (both originally authored by Armin Ronacher, the author of Flask).

In addition to the project website, look at the sources on GitHub, the PyPI entry, and the docs. To run Flask in Google App Engine (locally on your computer, or on Google's servers at *appspot.com*), see L. Henriquez and J. Euphrosine's GitHub page; or, for a somewhat richer and more structured approach, K. Gill's GitHub page.

The main class supplied by the `flask` package is named `Flask`. An instance of `flask.Flask`, besides being a WSGI application itself, also wraps a WSGI application as its `wsgi_app` property. When you need to further wrap the WSGI app in some WSGI middleware, use the idiom:

```
import flaskapp = flask.Flask(__name__)app.wsgi_app = some_middleware(app.wsgi_app)
```

When you instantiate `flask.Flask`, always pass it as the first argument the application name (often just the `__name__` special variable of the module where you instantiate it; if you instantiate it from within a package, usually in *__init__.py*, `__name__.partition('.')[0]` works). Optionally, you can also pass named parameters such as `static_folder` and `template_folder` to customize where static files and jinja2 templates are found; however, that's rarely needed—the default values (folders named *static* and *templates*, respectively, located in the same folder as the Python script that instantiates `flask.Flask`) make perfect sense.

An instance `app` of `flask.Flask` supplies more than 100 methods and properties, many of them decorators to bind functions to `app` in various roles, such as *view functions* (serving HTTP verbs on a URL) or *hooks* (letting you alter a request before it's processed, or a response after it's built; handling errors; and so forth).

`flask.Flask` takes few parameters at instantiation (and the ones it takes are not ones that you usually need to compute in your code), and it supplies decorators you'll want to use as you define, for example, view functions. Thus, the normal pattern in `flask` is to instantiate `app` early, just as your application is starting up, so that the app's decorators, and other methods and properties, are available as you `def` view functions and so on.

As there's a single global `app` object, you may wonder how thread-safe it can be to access, mutate, and rebind `app`'s properties and attributes. No worry: they're *proxies* to actual objects living in the *context* of a specific request, in a specific thread or greenlet. Never type-check those properties (their types are in fact obscure proxy types), and you'll be fine; type checking is not a good idea in Python, anyway.

Flask also supplies many other utility functions and classes; often, the latter subclass or wrap classes from other packages to add seamless, convenient Flask integration. For example, Flask's `Request` and `Response` classes add just a little handy functionality by subclassing the corresponding Werkzeug classes.

### Flask request objects

The class `flask.Request` supplies a large number of properties, thoroughly documented in Flask's docs. The ones you'll be using most often are:

`cookies`

A `dict` with the cookies from the request

`data`

A string, the request's body (for POST and PUT requests)

`files`

A `MultiDict` whose values are file-like objects, all files uploaded in the request (for POST and PUT requests), keys being the files' names

`headers`

A `MultiDict` with the request's headers

`values`

A `MultiDict` with the request's parameters (either from the query string or, for POST and PUT requests, from the form that's the request's body)

A `MultiDict` is like a `dict` except that it can have multiple values for a key. Indexing, and `get`, on a `MultiDict` instance *`m`*, return an arbitrary one of the values; to get the list of values for a key (an empty list, if the key is not in `m`), call *`m`*`.getlist(`*`key`*`)`.

### Flask response objects

Often, a Flask view function just returns a string (which becomes the response's body): Flask transparently wraps an instance *`r`* of `flask.Response` around the string, so you don't have to worry about the response class. However, sometimes you want to alter the response's headers; in this case, in the view function, call *`r`*`=flask.make_response(`*`astring`*`)`, alter `MultiDict` *`r`*`.headers` as you want, then `return` *`r`*. (To set a cookie, don't use *`r`*`.headers`; rather, call *`r`*`.set_cookie`, with arguments as mentioned in `set_cookie` in Table 20-1.)

Some of Flask's built-in integrations of other systems don't require subclassing: for example, the templating integration implicitly injects into the jinja2 context the Flask globals `config`, `request`, `session`, and `g` (the latter being the handy "globals catch-all" object `flask.g`, a proxy in application context, on which your code can store whatever you want to "stash" for the duration of the request being served), and the functions `url_for` (to translate an endpoint to the corresponding URL, same as `flask.url_for`) and `get_flashed_messages` (to support the advanced Flask concept of *flashed messages,* which we do not cover in this book; same as `flask.get_flashed_messages`). Flask provides convenient ways for your code to inject more filters, functions, and values into the jinja2 context, without any subclassing.

Most of the officially recognized or approved Flask extensions (over 50 at the time of this writing) adopt similar approaches, supplying classes and utility functions to seamlessly integrate other popular systems with your Flask applications.

Flask also introduces some advanced concepts of its own, such as signals to provide looser dynamic coupling in a “pub/sub” pattern, and blueprints, offering a substantial subset of a Flask application’s functionality to ease refactoring large applications in flexible ways. We do not cover these advanced concepts in this book.

Example 20-2 shows a simple Flask example.

### Example 20-2. A Flask example

```
import datetime, flask
app = flask.Flask(__name__)
app.permanent_session_lifetime = datetime.timedelta(days=365)
app.secret_key = b'\xc5\x8f\xbc\xa2\x1d\xeb\xb3\x94;:d\x03'

@app.route('/')
def greet():
    lastvisit = flask.session.get('lastvisit')
    now = datetime.datetime.now()
    newvisit = now.ctime()
    template = '''
      <html><head><title>Hello, visitor!
</title>
      </head>
<body>
       %  f lastvisit
{      i  %}
        <p>Welcome back to this site!
</p>
        <p>You last visited on {{lastvisit}}
UTC</p>
        <p>This visit on {{newvisit}}
UTC</p>
       %  lse
{      e  %}
        <p>Welcome to this site on your first visit!
</p>
        <p>This visit on {{newvisit}}
UTC</p>
        <p>Please Refresh the web page to
proceed</p>
       %
{      e  ndif %}
      </body>
</html>'''
    flask.session['lastvisit'] = newvisit
    return flask.render_template_string(
      template, newvisit=newvisit, lastvisit=lastvisit)
```

This Flask example just shows how to use some of the many building blocks that Flask offers—the `Flask` class, a view function, and rendering the response (in this case, using `render_template_string` on a jinja2 template; in real life, templates are usually kept in separate files rendered with `render_template`). The example also shows how to maintain continuity of state among multiple interactions with the server from the same browser, with the handy `flask.session` variable. (The example might alternatively have mimicked Example 20-1 more closely, putting together the HTML response in Python code instead of using jinja2, and using a cookie directly instead of the session; however, real-world Flask apps do tend to use jinja2 and sessions by preference.)

If this app had multiple view functions, no doubt it would want to set `lastvisit` in the session no matter through which URL the app was getting "visited." Here's how to code and decorate a hook function to execute after each request:

```
@app.after_request()
def set_lastvisit(response):
    now = datetime.datetime.now()
    flask.session['lastvisit'] = now.ctime
()
    return response
```

You can now remove the `flask.session['lastvisit'] = newvisit` statement from the view function `greet`, and the app will keep working fine.

## Bottle

Another worthy lightweight framework is Bottle. Bottle is implemented as a single *bottle.py* file (of over 4,000 lines!). It has no dependencies beyond the standard library. Strictly speaking, Bottle requires no installation process: you can just download it to your current directory—for example, with **`curl --location -O http://bottlepy.org/bottle.py`**, to get the latest development snapshot (although we recommend you install a stable release, as usual, with `pip install bottle`). Within that one *.py* file, Bottle supports both v2 and v3. It includes a web server useful for local development, and even a simple standalone template engine based on embedding Python in your templates (although it also lets you use jinja2 or another external template engine you have installed).

In addition to the project website, look at the source on GitHub. Bottle comes with adapters for many WSGI servers, including Google App Engine, but for a useful "skeleton" to run `bottle` on Google App Engine (locally on your computer, or on Google's servers at *appspot.com*), see L. Henriquez and J. Euphrosine's GitHub page.

The main class supplied by the `bottle` package is named `Bottle`. An instance of `bottle.Bottle`, besides being a WSGI application itself, also wraps a WSGI application as its `wsgi` attribute. When you need to further wrap the WSGI app in some WSGI middleware, use the idiom:

```
import bottleapp = bottle.Bottle()app.wsgi = some_middleware(app.wsgi)
```

When you instantiate `bottle.Bottle`, you can optionally pass the named parameters `catchall` and `autojson` (both default to `True`). When `catchall` is `True`, Bottle catches and handles all exceptions; pass it as `False` to let exceptions propagate (only useful when you wrap the WSGI app in some debugging middleware). When `autojson` is `True`, Bottle detects when a callback function returns a `dict`, and, in that case, serializes the `dict` with `json.dumps` as the response's body, and sets the response's `Content-Type` header to `'application/json'`; pass `autojson` as `False` to let such `dict`s propagate (only useful when you wrap the WSGI app in some middleware dealing with them).

An instance `app` of `bottle.Bottle` supplies dozens of methods and properties, including decorators to bind functions to `app` in various roles, such as *callback functions* (serving HTTP verbs on given URLs) or *hooks* (letting you alter a request before it's processed, or a response after it's built). The normal pattern in `bottle` is to instantiate `app` early, as your application is starting up, so the app's decorators, and other methods and properties, are available as you `def` callback functions and so on. Often, you don't even need to instantiate `bottle.Bottle` explicitly: Bottle makes a "default application object" for you, and module-level functions delegate to the default application's methods of the same name. When you want to access the default application object, call the function `bottle.default_app`. Relying on the default app is fine for small, simple web apps, although we recommend explicit instantiation as a more Pythonic style.

## Bottle routing

To route incoming requests to callback functions, Bottle supplies the function `route(`*`path`*`, method='GET', callback=None)`. You often use it as a decorator of the callback function, but, for flexibility, you also have the alternative of calling it directly and passing the callback function as the `callback` argument.

The `method` argument can be any HTTP method name ( `'GET'`, `'DELETE'`, `'PATCH'`, `'POST'`, `'PUT'`) or a list of HTTP method names. Bottle also supplies functions that are synonyms of `route` but with a specified value for `method`—these functions are named like each HTTP method, lowercased (`get`, `delete`, `patch`, `post`, `put`).

`path` is a string matching the URI to route; for example, `'/'` to route requests that ask for the root URL of the site. The string may include one or more *wildcards*—identifiers enclosed in angle brackets `<>`—in which case the callback function must accept arguments named like those identifiers (when a path has no wildcards, Bottle calls the callback function it routes to without arguments). Each wildcard matches one or more characters of the URI, up to the first slash in the URI, excluded.

For example, when `path` is `'/greet/<name>/first'`, and the URI string is `'/greet/alex/first'`, the callback function must accept an argument named `name`, and Bottle calls the function with value `'alex'` for that argument.

For even more flexibility, the wildcard's identifier can be followed by a *filter*. For any identifier `ident`: `<`*`ident`*`:int>` matches digits and converts them to an `int`; `<`*`ident`*`:float>` matches digits and an optional dot and converts them to a `float`; `<`*`ident`*`:path>` matches characters *including* slashes (in a nongreedy way, if further parts follow); and `<`*`ident`*`:re:`*`pattern`*`>` matches the arbitrary regular expression pattern you indicate. Bottle even lets you code your own custom filters, although we do not cover this advanced subject in this book.

## Bottle request objects

The `bottle.Request` class supplies a large number of properties, thoroughly documented in Bottle's docs. The ones you'll use most often are:

`body`

A seekable file-like object, the request's body (for POST and PUT requests)

`cookies`

A `FormsDict` with the cookies from the request `files`

> A `FormsDict` whose values are file-like objects, all files uploaded in the request (for POST and PUT requests), keys being the files' names

`headers`

> A `WSGIHeaderDict` with the request's headers

`params`

> A `FormsDict` with the request's parameters (either from the query string, or, for POST and PUT requests, from the form that's the body)

A `FormsDict` is like a `dict` except that it can have multiple values for a key. Indexing, and the method `get`, on a `FormsDict` instance *`f`*, return an arbitrary one of the values; to get the whole list of values for a key (possibly an empty list), call `f.getall(key)`. A `WSGIHeaderDict` is like a `dict` except that it has `str` keys and values (bytes in v2, Unicode in v3) and keys are case-insensitive.

*Signed* cookies in `bottle.request.cookies` are not decoded: call `bottle.request.get_cookie(key, secret)` (which returns `None` when the cookie is missing or was not signed with the given `secret` string) to properly access a signed cookie in decoded form.

`bottle.request` is a thread-safe proxy to the request being processed.

### Bottle response objects

Often, a Bottle callback function just returns a result: Bottle transparently wraps an instance *`r`* of `bottle.Response` around a bytestring body (prepared depending on the result's type), so you don't have to worry about the response class.

The result can be a bytestring (used as the body), a Unicode string (encoded as per `Content-Type`, `utf8` by default), a `dict` (serialized with `json.dumps`—this also sets the content type to `'application/json'`), a file-like object (anything with a `read` method), or any iterable whose items are (byte or Unicode) strings.

When the result is an instance of class `HTTPError` or `HTTPResponse`, this indicates an error, just like raising such an instance does (the difference between the two classes is that, when you return or raise an instance of `HTTPError`, Bottle applies an *error handler*, while instances of `HTTPResponse` bypass error handling).

When your callback function wants to alter the response's headers, it can do so on `bottle.response` before returning the value for the response's body.

`bottle.response` is a thread-safe proxy to the response being prepared.

To set a cookie, don't use `bottle.response.headers`; rather, call `bottle.response.set_cookie`, with arguments as mentioned in `set_cookie` in Table 20-1, plus, optionally, a named argument `secret=...`, whose value is used to cryptographically *sign* the cookie: signing makes the cookie safe from tampering (although it does not hide its contents), and Bottle takes advantage of that to let you use an object of any type as the cookie's value and serialize it with `pickle` (the `pickle.dumps` result bytestring must fit in the 4 KB cookie limit).

### Bottle templating

Bottle supplies its own built-in Simple Templating System, which we do not cover in this book. However, using other templating engines, such as jinja2 (covered in "The jinja2 Package") is also quite simple (provided that you have previously separately installed such engines in your Python installation or virtual environment).

By default, Bottle loads template files from either the current directory or its subdirectory *./views/* (you can change that by altering the list of strings `bottle.TEMPLATE_PATH`). To make a template object, call `bottle.template`: the first argument is either a template filename or a template string; every other positional argument is a `dict` cumulatively setting template variables, and so is every named argument, except, optionally, `template_adapter=...`, which lets you specify a template engine (defaulting to Bottle's own "simple templating"). `bottle.jinja2template` is the same function, but specifies the jinja2 adapter.

Once you have a template object, you can call its `render` method to obtain the strings it renders to—or, more often, you just return the template object from a callback function to have Bottle use the template object to build the response's body.

An even simpler way to use templating in a callback function is the decorator `bottle.view` ( `bottle.jinja2view`, to use jinja2), which takes the same arguments as `bottle.template`: when a callback function is decorated this way, and the function returns a `dict`, then that `dict` is used to set or override template variables, and Bottle proceeds as if the function had returned the template.

In other words, the following three functions are equivalent:

```
import bottle

@bottle.route('/a')
def a():
    t = bottle.template('tp.html', x=23
)
    return t.render()

@bottle.route('/b')
def b():
    t = bottle.template('tp.html', x=23
)
    return t

@bottle.route('/c')
@bottle.view('tp.html')
def c():
    return {'x': 23}
```

If Bottle's built-in functionality (plus all the extras you can build up for yourself by explicit use of standard library and third-party modules) isn't enough, Bottle also lets you extend its own functionality via *plug-ins*. Bottle officially recognizes and lists over a dozen reusable third-party plug-ins, and also documents how to write your own personal plug-ins. We don't cover Bottle plug-ins in this book.

Example 20-3 shows a simple Bottle example.

**Example 20-3. A Bottle example**

```
import datetime, bottle
one_year = datetime.timedelta(days=365)

@bottle.route('/')
def greet():
    lastvisit = bottle.request.get_cookie('lastvisit')
    now = datetime.datetime.now()
    newvisit = now.ctime()

    thisvisit = '<p>This visit on {}UTC</p>' .format(newvisit)
    bottle.response.set_cookie('lastvisit', newvisit,
                               expires=now+one_year)
                 '<html><head><title>Hello, visitor!
    resp_body = [</title>'
                 '</head><body>']
    if lastvisit is None:
        resp_body.extend((
          '<p>Welcome to this site on your first visit!
          </p>'                                              ,
          thisvisit,
          '<p>Please Refresh the web page to
          proceed</p>'                                    ))
    else:
        resp_body.extend((
          '<p>Welcome back to this site!
          </p>'                               ,

          '<p>You last visited on {}UTC</p>' .format(lastvisit
),
          thisvisit))
    resp_body.append('</body></html>')
    bottle.response.content_type = 'text/html'
    return resp_body

if __name__ == '__main__':
    bottle.run(debug=True)
```

This Bottle example just shows how to use some of the many building blocks that Bottle offers—the default app object, one callback function, and one route, preparing the response (in this case, just returning a list of string pieces of the response body, which is, as often, a handy way to build things up). The example also shows how to maintain a minimal continuity of state among multiple interactions with the server from the same browser by setting and using a cookie.

If this app had multiple routes, no doubt it would want to set the `lastvisit` cookie no matter through which route the app was getting "visited." Here's how to code and decorate a hook function to execute after each request:

```
@bottle.hook('after_request')
def set_lastvisit():
    now = datetime.datetime.now()
    bottle.response.set_cookie('lastvisit', now.ctime
(),
                               expires=now+one_year)
```

You can now remove the `bottle.response.set_cookie` call from the callback function `greet`, and the app will keep working fine.

## webapp2

Last, but not least, is webapp2, which shares with Django and jinja2 the distinction of being bundled with Google App Engine (but of course, just like jinja2 and Django, can also perfectly well be used with other web servers). To install webapp2 with Google App Engine, see the main Quick Start guide; to install it independently of Google App Engine, see the alternative Quick Start guide. You can also examine the source code and the PyPI entry.

The main class supplied by `webapp2` is named `WSGIApplication`. An instance of `webapp2.WSGIApplication` is a WSGI app, and you instantiate it with up to three named arguments: `routes`, a list of routing specifications; `debug`, a `bool` defaulting to `False` (pass as `True` to enable debugging mode); and `config`, a `dict` with configuration information for the application. Since `routes` is easier to express after you have defined the handlers you want to route to, the usual pattern in `webapp2` is to instantiate the WSGI application at the *end* of the module.

## webapp2 request objects

The `webapp2.Request` class (the current instance is available to your handler methods as `self.request`) supplies a large number of properties and methods, thoroughly documented in the webapp2 docs. The ones you'll be using most often are three mappings: `cookies`, cookie names to values; `headers`, header names to values (case-insensitive); and `params`, request parameter names to values. Two handy methods are `get` (like `params.get`, but you can optionally pass the named argument `allow_multiple=True` to get a list of all values for the parameter) and `get_all` (like `get` with `allow_multiple=True`).

## webapp2 response objects

The `webapp2.Response` class (the current instance is available to your handler methods as `self.response`) supplies a large number of properties and methods, thoroughly documented in the webapp2 docs. One you'll use often is the mapping property `headers`, which you use to set the response headers; however, for the specific purpose of setting cookies, use, instead, the method `set_cookie`. To set response status, assign to `status`—for example, `self.response.status = '404 Not Found'`.

To build up the response body, you normally call the response's `write` method (often more than once). Alternatively, assign to the attribute `body` a bytestring—or a Unicode string, which gets encoded as UTF-8, if the Content-Type header ends in `; charset=utf8` (or, equivalently, set the `charset` attribute to `'utf8'`).

## webapp2 handlers

webapp2 *handlers* are classes extending `webapp2.RequestHandler`. Handlers define methods named like (the lowercase version of) HTTP methods, such as `get` and `post`. Such methods access the current request as `self.request` and prepare the response as `self.response`.

A handler may optionally override the `dispatch` method, which gets called before the appropriate method (such as `get` or `post`) is determined. This lets the handler optionally alter the request, do custom dispatching, and also (typically after delegating to the superclass's `dispatch`) alter the resulting `self.response`. Such overriding is often done in a common base class: then, you inherit from that common base class in your actual handler classes.

## webapp2 routes

### webapp2 routes

webapp2 supplies a `Route` class for sophisticated routing needs. However, in the majority of cases, you can think of a *route* as just a pair `(`*`template`*`,`*`handler`*`)`.

The `template` is a string that may include identifiers enclosed in angle brackets `<>`. Such angle-bracket components match any sequence of characters (up to a slash, `/`, excluded) and pass the substring as a *named* argument to the handler's method.

Within the angle brackets, you might alternatively have a `:` (colon), followed by a regular expression pattern. In this case, the component matches characters per the pattern, and passes the substring as a *positional* argument to the handler's method.

Lastly, within the angle brackets, you may have an identifier, then a `:` (colon), then a regular expression pattern. In this case, the component matches characters per the pattern, and passes the substring as a *named* argument to the handler's method.

Positional arguments are passed only if no named ones are; if `template` has both kinds of components, the ones that would become positional arguments get matched, but their values are not passed at all to the handler's method. For example, if the template is `'/greet/<name>/<:.*>'`, the handler's method is called with named argument `name`, and without any positional argument corresponding to the trailing part of the URI.

Handler methods routed to by a route whose template includes angle-bracket components must accept the resulting positional or named parameters.

The `handler` is normally a *handler class* as covered in "webapp2 handlers". However, there are alternatives, of which the most frequently used one is to have `handler` be a *string* identifying the handler class, such as `'my.module.theclass'`. In this case, the relevant module is not imported until the handler class is specifically needed: if and when it's needed, webapp2 imports it on the fly (at most once).

### webapp2 extras

The webapp2.extras package supplies several convenient modules to seamlessly add functionality to webapp2, covering authentication and authorization, integration with jinja2 and other templating engines, i18n, and *sessions* (with various possible backing stores) for easier maintenance of state than with just cookies.

Example 20-4 shows a simple webapp2 example.

#### Example 20-4. A webapp2 example

```
import datetime, webapp2
one_year = datetime.timedelta(days=365)

class Greeter(webapp2.RequestHandler):
    def get(self):
        lastvisit = self.request.cookies.get('lastvisit')
        now = datetime.datetime.now()
        newvisit = now.ctime()

        thisvisit = '<p>This visit on {}UTC</p>' .format(newvisit)
        self.response.set_cookie('lastvisit', newvisit,
                                 expires=now+one_year)
        self.response.write(
            '<html><head><title>Hello, visitor!
            </title>'
            '</head><body>')
        if lastvisit is None:
            self.response.write(
              '<p>Welcome to this site on your first visit!
              </p>'                                          )
            self.response.write(thisvisit)
            self.response.write(
              '<p>Please Refresh the web page to
              proceed</p>'                                 )
        else:
            self.response.write(
              '<p>Welcome back to this site!
              </p>'                           )
            self.response.write(

              '<p>You last visited on {}UTC</p>' .format(lastvisit
))
            self.response.write(thisvisit)
        self.response.write('</body></html>')

app = webapp2.WSGIApplication(
    [('/', Greeter),
    ]
)
```

This webapp2 example just shows how to use some of the many building blocks that webapp2 offers—the app object, one handler, and one route, preparing the response (in this case, just writing to it). The example also shows how to maintain a minimal continuity of state among multiple interactions with the server from the same browser by setting and using a cookie.

If this app had multiple routes, no doubt it would want to set the `lastvisit` cookie no matter through which route the app was getting “visited.” Here’s how to code a base handler class (derive every handler from it!), have it override `dispatch`, and use the override to always set the needed cookie:

```
class BaseHandler(webapp2.RequestHandler):
    def dispatch(self):
        webapp2.RequestHandler.dispatch(self)
        now = datetime.datetime.now()
        self.response.set_cookie('lastvisit', now.ctime
(),
                                 expires=now+one_year)

class Greeter(BaseHandler): ...
```

You can now remove the `self.response.set_cookie` call from `Greeter.get`, and the app will keep working fine.