webapp2 Documentation

Release 2.1

Rodrigo Moraes

July 29, 2011

CONTENTS

webapp2 is a lightweight Python web framework compatible with Google App Engine's webapp.

webapp2 is a single file that follows the simplicity of webapp, but improves it in some ways: it extends webapp to offer better URI routing and exception handling, a full featured response object and a more flexible dispatching mechanism.

webapp2 also offers the package webapp2_extras with several optional utilities: sessions, localization, internationalization, domain and subdomain routing, secure cookies and support for threaded environments.

webapp2 can also be used outside of Google App Engine, independently of the App Engine SDK.

For a complete description of how webapp2 improves webapp, see *features*.

CHAPTER

ONE

QUICK LINKS

- Downloads
- Google Code Repository
- Discussion Group
- Download docs in PDF
- Follow us on Twitter

CHAPTER

TWO

TUTORIALS

2.1 Quick start

If you already know webapp, webapp2 is very easy to get started. You can use webapp2 exactly like webapp, and learn the new features as you go.

If you are new to App Engine, read *Getting Started with App Engine* first. You will need the App Engine SDK installed for this quick start.

Note: If you want to use webapp2 outside of App Engine, read the *Quick start (to use webapp2 outside of App Engine)* tutorial instead.

2.1.1 Create an application

Create a directory `hellowebapp2` for your new app. Download webapp2, unpack it and add `webapp2.py` to that directory. If you want to use extra features such as sessions, extra routes, localization, internationalization and more, also add the `webapp2_extras` directory to your app.

2.1.2 Hello, webapp2!

Create an `app.yaml` file in your app directory with the following contents:

```
application: hellowebapp2
version: 1
runtime: python
api_version: 1

handlers:
- url: /.*
  script: main.py
```

Then create a file `main.py` and define a handler to display a 'Hello, webapp2!' message:

```
import webapp2

class HelloWebapp2(webapp2.RequestHandler):
    def get(self):
        self.response.write('Hello, webapp2!')

app = webapp2.WSGIApplication([
```

```
    ('/', HelloWebapp2),
], debug=True)

def main():
    app.run()

if __name__ == '__main__':
    main()
```

2.1.3 Test your app

If you're using the Google App Engine Launcher, you can set up the application by selecting the **File** menu, **Add Existing Application...**, then selecting the `hellowebapp2` directory. Select the application in the app list, click the Run button to start the application, then click the Browse button to view it. Clicking Browse simply loads (or reloads) http://localhost:8080/ in your default web browser.

If you're not using Google App Engine Launcher, start the web server with the following command, giving it the path to the `hellowebapp2` directory:

```
google_appengine/dev_appserver.py helloworld/
```

The web server is now running, listening for requests on port 8080. You can test the application by visiting the following URL in your web browser:

http://localhost:8080/

2.2 Quick start (to use webapp2 outside of App Engine)

webapp2 can also be used outside of App Engine as a general purpose web framework, as it has these features:

- It is independent of the App Engine SDK. If the SDK is not found, it sets fallbacks to be used outside of App Engine.
- It supports threaded environments through the module *Local*.
- All webapp2_extras modules are designed to be thread-safe.
- It is compatible with `WebOb` 1.0 and superior, which fixes several bugs found in the version bundled with the SDK (which is of course supported as well).

It won't support App Engine services, but if you like webapp, why not use it in other servers as well? Here we'll describe how to do this.

Note: If you want to use webapp2 on App Engine, read the *Quick start* tutorial instead.

2.2.1 Prerequisites

If you don't have a package installer in your system yet (like `pip` or `easy_install`), install one. See *tutorials.installing.packages*.

If you don't have `virtualenv` installed in your system yet, install it. See *tutorials.virtualenv*.

2.2.2 Create a directory for your app

Create a directory `hellowebapp2` for your new app. It is where you will setup the environment and create your application.

2.2.3 Install WebOb, Paste and webapp2

We need three libraries to use webapp2: WebOb, for Request and Response objects, Paste, for the development server, and webapp2 itself.

Type this to install them using the **active virtual environment** (see *tutorials.virtualenv*):

```
$ pip install WebOb
$ pip install Paste
$ pip install webapp2
```

Or, using easy_install:

```
$ easy_install WebOb
$ easy_install Paste
$ easy_install webapp2
```

Now the environment is ready for your first app.

2.2.4 Hello, webapp2!

Create a file `main.py` inside your `hellowebapp2` directory and define a handler to display a 'Hello, webapp2!' message. This will be our bootstrap file:

```
import webapp2

class HelloWebapp2(webapp2.RequestHandler):
    def get(self):
        self.response.write('Hello, webapp2!')

app = webapp2.WSGIApplication([
    ('/', HelloWebapp2),
], debug=True)

def main():
    from paste import httpserver
    httpserver.serve(app, host='127.0.0.1', port='8080')

if __name__ == '__main__':
    main()
```

2.2.5 Test your app

Now start the development server using the Python executable provided by virtualenv:

```
$ python main.py
```

The web server is now running, listening for requests on port 8080. You can test the application by visiting the following URL in your web browser:

http://127.0.0.1:8080/

2.3 Getting Started with App Engine

This tutorial describes how to develop and deploy a simple Python project with Google App Engine. The example project, a guest book, demonstrates how to setup the Python runtime environment, how to use webapp2 and several App Engine services, including the datastore and the Google user service.

This tutorial has the following sections:

- *tutorials.gettingstarted.introduction*
- *tutorials.gettingstarted.devenvironment*
- *tutorials.gettingstarted.helloworld*
- *tutorials.gettingstarted.usingwebapp2*
- *tutorials.gettingstarted.usingusers*
- *tutorials.gettingstarted.handlingforms*
- *tutorials.gettingstarted.usingdatastore*
- *tutorials.gettingstarted.templates*
- *tutorials.gettingstarted.staticfiles*
- *tutorials.gettingstarted.uploading*

Note: This tutorial is a port from the official Python Getting Started guide from Google App Engine, created by the App Engine team and licensed under the Creative Commons Attribution 3.0 License.

2.4 Internationalization and localization with webapp2

In this tutorial we will learn how to get started with `webapp2_extras.i18n`. This module provides a complete collection of tools to localize and internationalize apps. Using it you can create applications adapted for different locales and timezones and with internationalized date, time, numbers, currencies and more.

2.4.1 Prerequisites

If you don't have a package installer in your system yet (like `pip` or `easy_install`), install one. See *tutorials.installing.packages*.

2.4.2 Get Babel and Pytz

The i18n module depends on two libraries: `babel` and `pytz` (or `gaepytz`). So before we start you must add the `babel` and `pytz` packages to your application directory (for App Engine) or install it in your virtual environment (for other servers).

For App Engine, download `babel` and `pytz` and add those libraries to your app directory:

- Babel: http://babel.edgewall.org/
- Pytz: http://pypi.python.org/pypi/gaepytz

For other servers, install those libraries in your system using `pip`. App Engine users also need babel installed, as we use the command line utility provided py it to extract and update message catalogs. This assumes a **nix* environment:

```
$ sudo pip install babel
$ sudo pip install gaepytz
```

Or, if you don't have pip but have `easy_install`:

```
$ sudo easy_install babel
$ sudo easy_install gaepytz
```

2.4.3 Create a directory for translations

We need a directory inside our app to store a messages catalog extracted from templates and Python files. Create a directory named `locale` for this.

If you want, later you can rename this directory the way you prefer and adapt the commands we describe below accordingly. If you do so, you must change the default i18n configuration to point to the right directory. The configuration is passed when you create an application, like this:

```
config = {}
config['webapp2_extras.i18n'] = {
    'translations_path': 'path/to/my/locale/directory',
}

app = webapp2.WSGIApplication(config=config)
```

If you use the default `locale` directory name, no configuration is needed.

2.4.4 Create a simple app to be translated

For the purposes of this tutorial we will create a very simple app with a single message to be translated. So create a new app and save this as `main.py`:

```
import webapp2

from webapp2_extras import i18n

class HelloWorldHandler(webapp2.RequestHandler):
    def get(self):
        # Set the requested locale.
        locale = self.request.GET.get('locale', 'en_US')
        i18n.get_i18n().set_locale(locale)

        message = i18n.gettext('Hello, world!')
        self.response.write(message)

app = webapp2.WSGIApplication([
    ('/', HelloWorldHandler),
], debug=True)

def main():
    app.run()

if __name__ == '__main__':
    main()
```

Any string that should be localized in your code and templates must be wrapped by the function `webapp2_extras.i18n.gettext()` (or the shortcut `_()`).

Translated strings defined in module globals or class definitions should use `webapp2_extras.i18n.lazy_gettext()` instead, because we want translations to be dynamic – if we call `gettext()` when the module is imported we'll set the value to a static translation for a given locale, and this is not what we want. `lazy_gettext()` solves this making the translation to be evaluated lazily, only when the string is used.

2.4.5 Extract and compile translations

We use the babel command line interface to extract, initialize, compile and update translations. Refer to Babel's manual for a complete description of the command options.

The extract command can extract not only messages from several template engines but also `gettext()` (from `gettext`) and its variants from Python files. Access your project directory using the command line and follow this quick how-to:

1. Extract all translations. We pass the current app directory to be scanned. This will create a `messages.pot` file in the `locale` directory with all translatable strings that were found:

```
$ pybabel extract -o ./locale/messages.pot ./
```

You can also provide a extraction mapping file that configures how messages are extracted. If the configuration file is saved as `babel.cfg`, we point to it when extracting the messages:

```
$ pybabel extract -F ./babel.cfg -o ./locale/messages.pot ./
```

2. Initialize the directory for each locale that your app will support. This is done only once per locale. It will use the `messages.pot` file created on step 1. Here we initialize three translations, `en_US`, `es_ES` and `pt_BR`:

```
$ pybabel init -l en_US -d ./locale -i ./locale/messages.pot
$ pybabel init -l es_ES -d ./locale -i ./locale/messages.pot
$ pybabel init -l pt_BR -d ./locale -i ./locale/messages.pot
```

3. Now the translation catalogs are created in the `locale` directory. Open each `.po` file and translate it. For the example above, we have only one message to translate: our `Hello, world!`.

Open `/locale/es_ES/LC_MESSAGES/messages.po` and translate it to `¡Hola, mundo!`.

Open `/locale/pt_BR/LC_MESSAGES/messages.po` and translate it to `Olá, mundo!`.

4. After all locales are translated, compile them with this command:

```
$ pybabel compile -f -d ./locale
```

That's it.

2.4.6 Update translations

When translations change, first repeat step 1 above. It will create a new `.pot` file with updated messages. Then update each locales:

```
$ pybabel update -l en_US -d ./locale/ -i ./locale/messages.pot
$ pybabel update -l es_ES -d ./locale/ -i ./locale/messages.pot
$ pybabel update -l pt_BR -d ./locale/ -i ./locale/messages.pot
```

After you translate the new strings to each language, repeat step 4, compiling the translations again.

2.4.7 Test your app

Start the development server pointing to the application you created for this tutorial and access the default language:

http://localhost:8080/

Then try the Spanish version:

http://localhost:8080/?locale=es_ES

And finally, try the Portuguese version:

http://localhost:8080/?locale=pt_BR

Voilà! Our tiny app is now available in three languages.

2.4.8 What else

The `webapp2_extras.i18n` module provides several other functionalities besides localization. You can use it to internationalize dates, currencies and numbers, and there are helpers to set the locale or timezone automatically for each request. Explore the API documentation to learn more.

CHAPTER

THREE

GUIDE

3.1 The WSGI application

The WSGI application receives requests and dispatches the appropriate handler, returning a response to the client. It stores the URI routes that the app will accept, configuration variables and registered objects that can be shared between requests. The WSGI app is also responsible for handling uncaught exceptions, avoiding that stack traces "leak" to the client when in production. Let's take an in depth look at it now.

Note: If the WSGI word looks totally unfamiliar to you, read the Another Do-It-Yourself Framework tutorial by Ian Bicking. It is a very recommended introduction to WSGI and you should at least take a quick look at the concepts, but following the whole tutorial is really worth. A more advanced reading is the WSGI specification described in the PEP 333.

3.1.1 Initialization

The `webapp2.WSGIApplication` class is initialized with three optional arguments:

- `routes`: a list of route definitions as described in *URI routing*.
- `debug`: a boolean flag that enables debug mode.
- `config`: a dictionary of configuration values for the application.

Compared to webapp, only config was added; it is used as a standard way to configure extra modules (sessions, internationalization, templates or your own app configuration values).

Everything is pretty straighforward:

```
import webapp2

routes = [
    (r'/', 'handlers.HelloWorldHandler'),
]

config = {}
config['webapp2_extras.sessions'] = {
    'secret_key': 'something-very-very-secret',
}

app = webapp2.WSGIApplication(routes=routes, debug=True, config=config)
```

3.1.2 Router

URI routing is a central piece in webapp2, and its main component is the `webapp2.Router` object, available in the application as the `webapp2.WSGIApplication.router` attribute.

The router object is responsible for everything related to mapping URIs to handlers. The router:

- Stores registered "routes", which map URIs to the application handlers that will handle those requests.
- Matches the current request against the registered routes and returns the handler to be used for that request (or raises a `HTTPNotFound` exception if no handler was found).
- Dispatches the matched handler, i.e., calling it and returning a response to the `WSGIApplication`.
- Builds URIs for the registered routes.

Using the `router` attribute you can, for example, add new routes to the application after initialization using the `add()` method:

```
import webapp2

app = webapp2.WSGIApplication()
app.router.add((r'/', 'handlers.HelloWorldHandler'))
```

The router has several methods to override how URIs are matched or built or how handlers are adapted or dispatched without even requiring subclassing. For an example of extending the default dispatching mechanism, see *Request handlers: returned values*.

Also check the Router API documentation for a description of the methods `webapp2.Router.set_matcher()`, `webapp2.Router.set_dispatcher()`, `webapp2.Router.set_adapter()` and `webapp2.Router.set_builder()`.

3.1.3 Config

When instantiating the app, you can pass a configuration dictionary which is then accessible through the `webapp2.WSGIApplication.config` attribute. A convention is to define configuration keys for each module, to avoid name clashes, but you can define them as you wish, really, unless the module requires a specific setup. First you define a configuration:

```
import webapp2

config = {'foo': 'bar'}

app = webapp2.WSGIApplication(routes=[
    (r'/', 'handlers.MyHandler'),
], config=config)
```

Then access it as you need. Inside a `RequestHandler`, for example:

```
import webapp2

class MyHandler(webapp2.RequestHandler):
    def get(self):
        foo = self.app.config.get('foo')
        self.response.write('foo value is %s' % foo)
```

3.1.4 Registry

A simple dictionary is available in the application to register instances that are shared between requests: it is the `webapp2.WSGIApplication.registry` attribute. It can be used by anything that your app requires and the intention is to avoid global variables in modules, so that you can have multiple app instances using different configurations: each app has its own extra instances for any kind of object that is shared between requests. A simple example that registers a fictitious `MyParser` instance if it is not yet registered:

```
import webapp2

def get_parser():
    app = webapp2.get_app()
    # Check if the instance is already registered.
    my_parser = app.registry.get('my_parser')
    if not my_parser:
        # Import the class lazily.
        cls = webapp2.import_string('my.module.MyParser')
        # Instantiate the imported class.
        my_parser = cls()
        # Register the instance in the registry.
        app.registry['my_parser'] = my_parser

    return my_parser
```

The registry can be used to lazily instantiate objects when needed, and keep a reference in the application to be reused.

A registry dictionary is also available in the *request object*, to store shared objects used during a single request.

3.1.5 Error handlers

As described in *Exception handling*, a dictionary is available in the app to register error handlers as the `webapp2.WSGIApplication.error_handlers` attribute. They will be used as a last resource if exceptions are not caught by handlers. It is a good idea to set at least error handlers for 404 and 500 status codes:

```
import logging

import webapp2

def handle_404(request, response, exception):
    logging.exception(exception)
    response.write('Oops! I could swear this page was here!')
    response.set_status(404)

def handle_500(request, response, exception):
    logging.exception(exception)
    response.write('A server error occurred!')
    response.set_status(500)

app = webapp2.WSGIApplication([
    webapp2.Route(r'/', handler='handlers.HomeHandler', name='home')
])
app.error_handlers[404] = handle_404
app.error_handlers[500] = handle_500
```

3.1.6 Debug flag

A debug flag is passed to the WSGI application on instantiation and is available as the `webapp2.WSGIApplication.debug` attribute. When in debug mode, any exception that is now caught is raised and the stack trace is displayed to the client, which helps debugging. When not in debug mode, a '500 Internal Server Error' is displayed instead.

You can use that flag to set special behaviors for the application during development.

For App Engine, it is possible to detect if the code is running using the SDK or in production checking the 'SERVER_SOFTWARE' environ variable:

```
import os

import webapp2

debug = os.environ.get('SERVER_SOFTWARE', '').startswith('Dev')

app = webapp2.WSGIApplication(routes=[
    (r'/', 'handlers.HelloWorldHandler'),
], debug=debug)
```

3.1.7 Thread-safe application

By default, webapp2 is thread-safe when the module `webapp2_extras.local` is available. This means that it can be used outside of App Engine or in the upcoming App Engine Python 2.7 runtime. This also works in non-threaded environments such as App Engine Python 2.5.

See in the *Quick start (to use webapp2 outside of App Engine)* tutorial an explanation on how to use webapp2 outside of App Engine.

3.1.8 Running the app

The application is executed in a CGI environment using the method `webapp2.WSGIApplication.run()`. When using App Engine, it uses the functions `run_bare_wsgi_app` or `run_wsgi_app` from `google.appengine.ext.webapp.util`. Outside of App Engine, it uses the `wsgiref.handlers` module. Here's the simplest example:

```
import webapp2

class HelloWebapp2(webapp2.RequestHandler):
    def get(self):
        self.response.write('Hello, webapp2!')

app = webapp2.WSGIApplication([
    ('/', HelloWebapp2),
], debug=True)

def main():
    app.run()

if __name__ == '__main__':
    main()
```

3.1.9 Unit testing

As described in *Unit testing*, the application has a convenience method to test handlers: `webapp2.WSGIApplication.get_response()`. It receives the same parameters as `Request.blank()` to build a request and call the application, returning the resulting response from a handler:

```
class HelloHandler(webapp2.RequestHandler):
    def get(self):
        self.response.write('Hello, world!')

app = webapp2.WSGIapplication([('/', HelloHandler)])

# Test the app, passing parameters to build a request.
response = app.get_response('/')
assert response.status_int == 200
assert response.body == 'Hello, world!'
```

3.1.10 Getting the current app

The active `WSGIApplication` instance can be accessed at any place of your app using the function `webapp2.get_app()`. This is useful, for example, to access the app registry or configuration values:

```
import webapp2

app = webapp2.get_app()
config_value = app.config.get('my-config-key')
```

3.2 URI routing

URI routing is the process of taking the requested URI and deciding which application handler will handle the current request. For this, we initialize the `WSGIApplication` defining a list of *routes*: each *route* analyses the current request and, if it matches certain criterias, returns the handler and optional variables extracted from the URI.

webapp2 offers a powerful and extensible system to match and build URIs, which is explained in details in this section.

3.2.1 Simple routes

The simplest form of URI route in webapp2 is a tuple `(regex, handler)`, where *regex* is a regular expression to match the requested URI path and *handler* is a callable to handle the request. This routing mechanism is fully compatible with App Engine's webapp framework.

This is how it works: a list of routes is registered in the *WSGI application*. When the application receives a request, it tries to match each one in order until one matches, and then call the corresponding handler. Here, for example, we define three handlers and register three routes that point to those handlers:

```
class HomeHandler(webapp2.RequestHandler):
    def get(self):
        self.response.write('This is the HomeHandler.')

class ProductListHandler(webapp2.RequestHandler):
    def get(self):
        self.response.write('This is the ProductListHandler.')

class ProductHandler(webapp2.RequestHandler):
```

```
    def get(self, product_id):
        self.response.write('This is the ProductHandler. '
            'The product id is %s' % product_id)

app = webapp2.WSGIApplication([
    (r'/', HomeHandler),
    (r'/products', ProductListHandler),
    (r'/products/(\d+)', ProductHandler),
])
```

When a request comes in, the application will match the request path to find the corresponding handler. If no route matches, an `HTTPException` is raised with status code 404, and the WSGI application can handle it accordingly (see *Exception handling*).

The *regex* part is an ordinary regular expression (see the `re` module) that can define groups inside parentheses. The matched group values are passed to the handler as positional arguments. In the example above, the last route defines a group, so the handler will receive the matched value when the route matches (one or more digits in this case).

The *handler* part is a callable as explained in *Request handlers*, and can also be a string in dotted notation to be lazily imported when needed (see explanation below in *Lazy Handlers*).

Simple routes are easy to use and enough for a lot of cases but don't support keyword arguments, URI building, domain and subdomain matching, automatic redirection and other useful features. For this, webapp2 offers the extended routing mechanism that we'll see next.

3.2.2 Extended routes

webapp2 introduces a routing mechanism that extends the webapp model to provide additional features:

- **URI building:** the registered routes can be built when needed, avoiding hardcoded URIs in the app code and templates. If you change the route definition in a compatible way during development, all places that use that route will continue to point to the correct URI. This is less error prone and easier to maintain.
- **Keyword arguments:** handlers can receive keyword arguments from the matched URIs. This is easier to use and also more maintanable than positional arguments.
- **Nested routes:** routes can be extended to match more than the request path. We will see below a route class that can also match domains and subdomains.

And several other features and benefits.

The concept is similar to the simple routes we saw before, but instead of a tuple `(regex, handler)`, we define each route using the class `webapp2.Route`. Let's remake our previous routes using it:

```
app = webapp2.WSGIApplication([
    webapp2.Route(r'/', handler=HomeHandler, name='home'),
    webapp2.Route(r'/products', handler=ProductListHandler, name='product-list'),
    webapp2.Route(r'/products/<product_id:\d+>', handler=ProductHandler, name='product'),
])
```

The first argument in the routes above is a *regex template*, the *handler* argument is the *request handler* to be used, and the *name* argument third is a name used to *build a URI* for that route.

Check `webapp2.Route.__init__()` in the API reference for the parameters accepted by the `Route` constructor. We will explain some of them in details below.

The regex template

The regex template can have variables enclosed by <> that define a name, a regular expression or both. Examples:

Format	Example
<name>	'/blog/<year>/<month>'
<:regex>	'/blog/<:\d{4}>/<:\d{2}>'
<name:regex>	'/blog/<year:\d{4}>/<month:\d{2}>'

The same template can mix parts with name, regular expression or both.

If the name is set, the value of the matched regular expression is passed as keyword argument to the handler. Otherwise it is passed as positional argument.

If only the name is set, it will match anything except a slash. So these routes are equivalent:

```
Route('/<user_id>/settings', handler=SettingsHandler, name='user-settings')
Route('/<user_id:[^/]+>/settings', handler=SettingsHandler, name='user-settings')
```

Note: The handler only receives `*args` if no named variables are set. Otherwise, the handler only receives `**kwargs`. This allows you to set regular expressions that are not captured: just mix named and unnamed variables and the handler will only receive the named ones.

Lazy handlers

One additional feature compared to webapp is that the handler can also be defined as a string in dotted notation to be lazily imported when needed.

This is useful to avoid loading all modules when the app is initialized: we can define handlers in different modules without needing to import all of them to initialize the app. This is not only convenient but also speeds up the application startup.

The string must contain the package or module name and the name of the handler (a class or function name). Our previous example could be rewritten using strings instead of handler classes and splitting our handlers in two files, `handlers.py` and `products.py`:

```
app = webapp2.WSGIApplication([
    (r'/', 'handlers.HomeHandler'),
    (r'/products', 'products.ProductListHandler'),
    (r'/products/(\d+)', 'products.ProductHandler'),
])
```

In the first time that one of these routes matches, the handlers will be automatically imported by the routing system.

Custom methods

A parameter `handler_method` can define the method of the handler that will be called, if handler is a class. If not defined, the default behavior is to translate the HTTP method to a handler method, as explained in *Request handlers*. For example:

```
webapp2.Route(r'/products', handler='handlers.ProductsHandler', name='products-list', handler_method=
```

Alternatively, the handler method can be defined in the handler string, separated by a colon. This is equivalent to the previous example:

```
webapp2.Route(r'/products', handler='handlers.ProductsHandler:list_products', name='products-list')
```

Restricting HTTP methods

If needed, the route can define a sequence of allowed HTTP methods. Only if the request method is in that list or tuple the route will match. If the method is not allowed, an `HTTPMethodNotAllowed` exception is raised with status code 405. For example:

```
webapp2.Route(r'/products', handler='handlers.ProductsHandler', name='products-list', methods=['GET'
```

This is useful when using functions as handlers, or alternative handlers that don't translate the HTTP method to the handler method like the default `webapp2.RequestHandler` does.

Restricting URI schemes

Like with HTTP methods, you can specify the URI schemes allowed for a route, if needed. This is useful if some URIs must be accessed using 'http' or 'https' only. For this, set the `schemes` parameter when defining a route:

```
webapp2.Route(r'/products', handler='handlers.ProductsHandler', name='products-list', schemes=['https
```

The above route will only match if the URI scheme is 'https'.

3.2.3 Domain and subdomain routing

The routing system can also handle domain and subdomain matching. This is done using a special route class provided in the `webapp2_extras.routes` module: the `webapp2_extras.routes.DomainRoute`. It is initialized with a pattern to match the current server name and a list of nested `webapp2.Route` instances that will only be tested if the domain or subdomain matches.

For example, to restrict routes to a subdomain of the appspot domain:

```
import webapp2
from webapp2_extras import routes

app = webapp2.WSGIApplication([
    routes.DomainRoute('<subdomain>.app-id.appspot.com', [
        webapp2.Route('/', handler=SubdomainHomeHandler, name='subdomain-home'),
    ]),
    webapp2.Route('/', handler=HomeHandler, name='home'),
])
```

In the example above, we define a template `'<subdomain>.app-id.appspot.com'` for the domain matching. When a request comes in, only if the request server name matches that pattern, the nested route will be tested. Otherwise the routing system will test the next route until one matches. So the first route `/foo` will only match when a subdomain of the `app-id.appspot.com` domain is accessed: when a subdomain is accessed and the path is `/`, the handler `SubdomainHomeHandler` will be used, but when no subdomain is accessed (or the domain is different) the `HomeHandler` will be used instead.

The template follows the same syntax used by `webapp2.Route` and must define named groups if any value must be added to the match results. In the example above, an extra *subdomain* keyword is passed to the handler, but if the regex didn't define any named groups, nothing would be added.

Matching only www, or anything except www

A common need is to set some routes for the main subdomain (`www`) and different routes for other submains. The webapp2 routing system can handle this easily.

To match only the `www` subdomain, simple set the domain template to a fixed value:

```
routes.DomainRoute('www.mydomain.com', [
    webapp2.Route('/', handler=HomeHandler, name='home'),
])
```

To match any subdomain except the `www` subdomain, set a regular expression that excludes `www`:

```
routes.DomainRoute(r'<subdomain:(?!www\.)[^.]+>.mydomain.com', [
    webapp2.Route('/', handler=HomeHandler, name='home'),
])
```

Any subdomain that matches and is not `www` will be passed as a parameter `subdomain` to the handler.

Similarly, you can restrict matches to the main `appspot` domain **or** a `www` domain from a custom domain:

```
routes.DomainRoute(r'<:(app-id\.appspot\.com|www\.mydomain\.com)>', [
    webapp2.Route('/', handler=HomeHandler, name='home'),
])
```

And then have a route that matches subdomains of the main `appspot` domain **or** from a custom domain, except `www`:

```
routes.DomainRoute(r'<subdomain:(?!www)[^.]+>.<:(app-id\.appspot\.com|mydomain\.com)>', [
    webapp2.Route('/', handler=HomeHandler, name='home'),
])
```

3.2.4 Path prefix routes

The `webapp2_extras.routes` provides a class to wrap routes that start with a common path: the `webapp2_extras.routes.PathPrefixRoute`. The intention is to avoid repetition when defining routes.

For example, imagine we have these routes:

```
app = WSGIApplication([
    Route('/users/<user:\w+>/', UserOverviewHandler, 'user-overview'),
    Route('/users/<user:\w+>/profile', UserProfileHandler, 'user-profile'),
    Route('/users/<user:\w+>/projects', UserProjectsHandler, 'user-projects'),
])
```

We could refactor them to reuse the common path prefix:

```
import webapp2
from webapp2_extras import routes

app = WSGIApplication([
    routes.PathPrefixRoute('/users/<user:\w+>', [
        webapp2.Route('/', UserOverviewHandler, 'user-overview'),
        webapp2.Route('/profile', UserProfileHandler, 'user-profile'),
        webapp2.Route('/projects', UserProjectsHandler, 'user-projects'),
    ]),
])
```

This is not only convenient, but also performs better: the nested routes will only be tested if the path prefix matches.

3.2.5 Other prefix routes

The `webapp2_extras.routes` has other convenience classes that accept nested routes with a common attribute prefix:

- webapp2_extras.routes.HandlerPrefixRoute: receives a handler module prefix in dotted notation and a list of routes that use that module.
- webapp2_extras.routes.NamePrefixRoute: receives a handler name prefix and a list of routes that start with that name.

3.2.6 Building URIs

Because our routes have a `name`, we can use the routing system to build URIs whenever we need to reference those resources inside the application. This is done using the function `webapp2.uri_for()` or the method `webapp2.RequestHandler.uri_for()` inside a handler, or calling `webapp2.Router.build()` directly (a `Router` instance is set as an attribute `router` in the WSGI application).

For example, if you have these routes defined for the application:

```
app = webapp2.WSGIApplication([
    webapp2.Route('/', handler='handlers.HomeHandler', name='home'),
    webapp2.Route('/wiki', handler=WikiHandler, name='wiki'),
    webapp2.Route('/wiki/<page>', handler=WikiHandler, name='wiki-page'),
])
```

Here are some examples of how to generate URIs for them:

```
# /
uri = uri_for('home')
# http://localhost:8080/
uri = uri_for('home', _full=True)
# /wiki
uri = uri_for('wiki')
# http://localhost:8080/wiki
uri = uri_for('wiki', _full=True)
# http://localhost:8080/wiki#my-heading
uri = uri_for('wiki', _full=True, _fragment='my-heading')
# /wiki/my-first-page
uri = uri_for('wiki-page', page='my-first-page')
# /wiki/my-first-page?format=atom
uri = uri_for('wiki-page', page='my-first-page', format='atom')
```

Variables are passed as positional or keyword arguments and are required if the route defines them. Keyword arguments that are not present in the route are added to the URI as a query string.

Also, when calling `uri_for()`, a few keywords have special meaning:

_full If True, builds an absolute URI.

_scheme URI scheme, e.g., *http* or *https*. If defined, an absolute URI is always returned.

_netloc Network location, e.g., *www.google.com*. If defined, an absolute URI is always returned.

_fragment If set, appends a fragment (or "anchor") to the generated URI.

Check `webapp2.Router.build()` in the API reference for a complete explanation of the parameters used to build URIs.

3.2.7 Routing attributes in the request object

The parameters from the matched route are set as attributes of the request object when a route matches. They are `request.route_args`, for positional arguments, and `request.route_kwargs`, for keyword arguments.

The matched route object is also available as `request.route`.

3.3 Request handlers

In the webapp2 vocabulary, *request handler* or simply *handler* is a common term that refers to the callable that contains the application logic to handle a request. This sounds a lot abstract, but we will explain everything in details in this section.

3.3.1 Handlers 101

A handler is equivalent to the *Controller* in the MVC terminology: in a simplified manner, it is where you process the request, manipulate data and define a response to be returned to the client: HTML, JSON, XML, files or whatever the app requires.

Normally a handler is a class that extends `webapp2.RequestHandler` or, for compatibility purposes, `webapp.RequestHandler`. Here is a simple one:

```
class ProductHandler(webapp2.RequestHandler):
    def get(self, product_id):
        self.response.write('You requested product %r.' % product_id)

app = webapp2.WSGIApplication([
    (r'/products/(\d+)', ProductHandler),
])
```

This code defines one request handler, `ProductHandler`, and a WSGI application that maps the URI `r'/products/(\d+)'` to that handler. When the application receives an HTTP request to a path that matches this regular expression, it instantiates the handler and calls the corresponding HTTP method from it. The handler above can only handle `GET` HTTP requests, as it only defines a `get()` method. To handle `POST` requests, it would need to implement a `post()` method, and so on.

The handler method receives a `product_id` extracted from the URI, and sets a simple message containing the id as response. Not very useful, but this is just to show how it works. In a more complete example, the handler would fetch a corresponding record from a database and set an appropriate response – HTML, JSON or XML with details about the requested product, for example.

For more details about how URI variables are defined, see *URI routing*.

3.3.2 HTTP methods translated to class methods

The default behavior of the `webapp2.RequestHandler` is to call a method that corresponds with the HTTP action of the request, such as the `get()` method for a HTTP GET request. The method processes the request and prepares a response, then returns. Finally, the application sends the response to the client.

The following example defines a request handler that responds to HTTP GET requests:

```
class AddTwoNumbers(webapp2.RequestHandler):
    def get(self):
        try:
            first = int(self.request.get('first'))
            second = int(self.request.get('second'))

            self.response.write("<html><body><p>%d + %d = %d</p></body></html>" %
                                (first, second, first + second))
        except (TypeError, ValueError):
            self.response.write("<html><body><p>Invalid inputs</p></body></html>")
```

A request handler can define any of the following methods to handle the corresponding HTTP actions:

- `get()`
- `post()`
- `head()`
- `options()`
- `put()`
- `delete()`
- `trace()`

3.3.3 View functions

In some Python frameworks, handlers are called *view functions* or simply *views*. In Django, for example, *views* are normally simple functions that handle a request. Our examples use mostly classes, but webapp2 handlers can also be normal functions equivalent to Django's *views*.

A webapp2 handler can, really, be **any** callable. The routing system has hooks to adapt how handlers are called, and two default adapters are used whether it is a function or a class. So, differently from webapp, ordinary functions can easily be used to handle requests in webapp2, and not only classes. The following example demonstrates it:

```
def display_product(request, *args, **kwargs):
    return webapp2.Response('You requested product %r.' % args[0])

app = webapp2.WSGIApplication([
    (r'/products/(\d+)', display_product),
])
```

Here, our handler is a simple function that receives the request instance, positional route variables as `*args` and named variables as `**kwargs`, if they are defined.

Apps can have mixed handler classes and functions. Also it is possible to implement new interfaces to define how handlers are called: this is done setting new handler adapters in the routing system.

Functions are an alternative for those that prefer their simplicity or think that handlers don't benefit that much from the power and flexibility provided by classes: inheritance, attributes, grouped methods, descriptors, metaclasses, etc. An app can have mixed handler classes and functions.

Note: We avoid using the term *view* because it is often confused with the *View* definition from the classic *MVC* pattern. Django prefers to call its *MVC* implementation *MTV* (model-template-view), so *view* may make sense in their terminology. Still, we think that the term can cause unnecessary confusion and prefer to use *handler* instead, like in other Python frameworks (webapp, web.py or Tornado, for instance). In essence, though, they are synonyms.

3.3.4 Returned values

A handler method doesn't need to return anything: it can simply write to the response object using `self.response.write()`.

But a handler **can** return values to be used in the response. Using the default dispatcher implementation, if a handler returns anything that is not `None` it **must** be a `webapp2.Response` instance. If it does so, that response object is used instead of the default one.

For example, let's return a response object with a *Hello, world* message:

```
class HelloHandler(webapp2.RequestHandler):
    def get(self):
        return webapp2.Response('Hello, world!')
```

This is the same as:

```
class HelloHandler(webapp2.RequestHandler):
    def get(self):
        self.response.write('Hello, world!')
```

What if you think that returning a response object is verbose, and want to return simple strings? Fortunately webapp2 has all the necessary hooks to make this possible. To achieve it, we need to extend the router dispatcher to build a `Response` object using the returned string. We can go even further and also accept tuples: if a tuple is returned, we use its values as positional arguments to instantiate the `Response` object. So let's define our custom dispatcher and a handler that returns a string:

```
def custom_dispatcher(router, request, response):
    rv = router.default_dispatcher(request, response)
    if isinstance(rv, basestring):
        rv = webapp2.Response(rv)
    elif isinstance(rv, tuple):
        rv = webapp2.Response(*rv)

    return rv

class HelloHandler(webapp2.RequestHandler):
    def get(self, *args, **kwargs):
        return 'Hello, world!'

app = webapp2.WSGIApplication([
    (r'/', HelloHandler),
])
app.router.set_dispatcher(custom_dispatcher)
```

And that's all. Now we have a custom dispatcher set using the router method `webapp2.Router.set_dispatcher()`. Our `HelloHandler` returns a string (or it could be tuple) that is used to create a `Response` object.

Our custom dispatcher could implement its own URI matching and handler dispatching mechanisms from scratch, but in this case it just extends the default dispatcher a little bit, wrapping the returned value under certain conditions.

3.3.5 A micro-framework based on webapp2

Following the previous idea of a custom dispatcher, we could go a little further and extend webapp2 to accept routes registered using a decorator, like in those Python micro-frameworks.

Without much ado, ladies and gentlemen, we present micro-webapp2:

```
import webapp2

class WSGIApplication(webapp2.WSGIApplication):
    def __init__(self, *args, **kwargs):
        super(WSGIApplication, self).__init__(*args, **kwargs)
        self.router.set_dispatcher(self.__class__.custom_dispatcher)

    @staticmethod
    def custom_dispatcher(router, request, response):
        rv = router.default_dispatcher(request, response)
```

```
        if isinstance(rv, basestring):
            rv = webapp2.Response(rv)
        elif isinstance(rv, tuple):
            rv = webapp2.Response(*rv)

        return rv

    def route(self, *args, **kwargs):
        def wrapper(func):
            self.router.add(webapp2.Route(handler=func, *args, **kwargs))
            return func

        return wrapper
```

Save the above code as `micro_webapp2.py`. Then you can import it in `main.py` and define your handlers and routes like this:

```
import micro_webapp2

app = micro_webapp2.WSGIApplication()

@app.route('/')
def hello_handler(request, *args, **kwargs):
    return 'Hello, world!'

def main():
    app.run()

if __name__ == '__main__':
    main()
```

This example just demonstrates some of the power and flexibility that lies behind webapp2; explore the *webapp2 API* to discover other ways to modify or extend the application behavior.

3.3.6 Overriding __init__()

If you want to override the `webapp2.RequestHandler.__init__()` method, you must call `webapp2.RequestHandler.initialize()` at the beginning of the method. It'll set the current request, response and app objects as attributes of the handler. For example:

```
class MyHandler(webapp2.RequestHandler):
    def __init__(self, request, response):
        # Set self.request, self.response and self.app.
        self.initialize(request, response)

        # ... add your custom initializations here ...
        # ...
```

3.3.7 Overriding dispatch()

One of the advantadges of webapp2 over webapp is that you can wrap the dispatching process of `webapp2.RequestHandler` to perform actions before and/or after the requested method is dispatched. You can do this overriding the `webapp2.RequestHandler.dispatch()` method. This can be useful, for example, to test if requirements were met before actually dispatching the requested method, or to perform actions in the response object after the method was dispatched. Here's an example:

```
class MyHandler(webapp2.RequestHandler):
    def dispatch(self):
        # ... check if requirements were met ...
        # ...

        if requirements_were_met:
            # Parent class will call the method to be dispatched
            # -- get() or post() or etc.
            super(MyHandler, self).dispatch()
        else:
            self.abort(403)
```

In this case, if the requirements were not met, the method won't ever be dispatched and a "403 Forbidden" response will be returned instead.

There are several possibilities to explore overriding `dispatch()`, like performing common checkings, setting common attributes or post-processing the response.

3.4 Request data

The request handler instance can access the request data using its `request` property. This is initialized to a populated WebOb `Request` object by the application.

The request object provides a `get()` method that returns values for arguments parsed from the query and from POST data. The method takes the argument name as its first parameter. For example:

```
class MyHandler(webapp2.RequestHandler):
    def post(self):
        name = self.request.get('name')
```

By default, `get()` returns the empty string (") if the requested argument is not in the request. If the parameter `default_value` is specified, `get()` returns the value of that parameter instead of the empty string if the argument is not present.

If the argument appears more than once in a request, by default `get()` returns the first occurrence. To get all occurrences of an argument that might appear more than once as a list (possibly empty), give `get()` the argument `allow_multiple=True`:

```
# <input name="name" type="text" />
name = self.request.get("name")

# <input name="subscribe" type="checkbox" value="yes" />
subscribe_to_newsletter = self.request.get("subscribe", default_value="no")

# <select name="favorite_foods" multiple="true">...</select>
favorite_foods = self.request.get("favorite_foods", allow_multiple=True)

# for food in favorite_foods:
# ...
```

For requests with body content that is not a set of CGI parameters, such as the body of an HTTP PUT request, the request object provides the attributes `body` and `body_file`: `body` is the body content as a byte string and `body_file` provides a file-like interface to the same data:

```
uploaded_file = self.request.body
```

3.4.1 GET data

Query string variables are available in `request.GET` or `request.str_GET`. Both carry the same values, but in the first they are converted to unicode, and in the latter they are strings.

`GET` or `str_GET` are a MultiDict: they act as a dictionary but the same key can have multiple values. When you call `.get(key)` for a key that has multiple values, the last value is returned. To get all values for a key, use `.getall(key)`. Examples:

```
request = Request.blank('/test?check=a&check=b&name=Bob')

# The whole MultiDict:
# GET([('check', 'a'), ('check', 'b'), ('name', 'Bob')])
get_values = request.str_GET

# The last value for a key: 'b'
check_value = request.str_GET['check']

# All values for a key: ['a', 'b']
check_values = request.str_GET.getall('check')

# An iterable with alll items in the MultiDict:
# [('check', 'a'), ('check', 'b'), ('name', 'Bob')]
request.str_GET.items()
```

The name `GET` is a bit misleading, but has historical reasons: `request.GET` is not only available when the HTTP method is GET. It is available for any request with query strings in the URI, for any HTTP method: GET, POST, PUT etc.

3.4.2 POST data

Variables url encoded in the body of a request (generally a POST form submitted using the `application/x-www-form-urlencoded` media type) are available in `request.POST` or `request.str_POST` (the first as unicode and the latter as string).

Like `request.GET` and `request.str_GET`, they are a MultiDict and can be accessed in the same way. Examples:

```
request = Request.blank('/')
request.method = 'POST'
request.body = 'check=a&check=b&name=Bob'

# The whole MultiDict:
# POST([('check', 'a'), ('check', 'b'), ('name', 'Bob')])
post_values = request.str_POST

# The last value for a key: 'b'
check_value = request.str_POST['check']

# All values for a key: ['a', 'b']
check_values = request.str_POST.getall('check')

# An iterable with alll items in the MultiDict:
# [('check', 'a'), ('check', 'b'), ('name', 'Bob')]
request.str_POST.items()
```

Like `GET`, the name `POST` is a bit misleading, but has historical reasons: they are also available when the HTTP method is PUT, and not only POST.

3.4.3 GET + POST data

`request.params` combines the variables from `GET` and `POST`. It can be used when you don't care where the variable comes from.

3.4.4 Files

Uploaded files are available as `cgi.FieldStorage` (see the `cgi` module) instances directly in `request.POST`.

3.4.5 Cookies

Cookies can be accessed in `request.cookies`. It is a simple dictionary:

```
request = Request.blank('/')
request.headers['Cookie'] = 'test=value'

# A value: 'value'
cookie_value = request.cookies.get('test')
```

See Also:

How to set cookies using the response object

3.4.6 Common Request attributes

body A file-like object that gives the body of the request.

content_type Content-type of the request body.

method The HTTP method, e.g., 'GET' or 'POST'.

url Full URI, e.g., `'http://localhost/blog/article?id=1'`.

scheme URI scheme, e.g., 'http' or 'https'.

host URI host, e.g., `'localhost:80'`.

host_url URI host including scheme, e.g., `'http://localhost'`.

path_url URI host including scheme and path, e.g., `'http://localhost/blog/article'`.

path URI path, e.g., `'/blog/article'`.

path_qs URI path including the query string, e.g., `'/blog/article?id=1'`.

query_string Query string, e.g., `id=1`.

headers A dictionary like object with request headers. Keys are case-insensitive.

GET A dictionary-like object with variables from the query string, as unicode.

str_GET A dictionary-like object with variables from the query string, as a string.

POST A dictionary-like object with variables from a POST form, as unicode.

str_POST A dictionary-like object with variables from a POST form, as a strings.

params A dictionary-like object combining the variables GET and POST.

cookies A dictionary-like object with cookie values.

3.4.7 Extra attributes

The parameters from the matched `webapp2.Route` are set as attributes of the request object. They are `request.route_args`, for positional arguments, and `request.route_kwargs`, for keyword arguments. The matched route object is available as `request.route`.

A reference to the active WSGI application is also set as an attribute of the request. You can access it in `request.app`.

3.4.8 Getting the current request

The active `Request` instance can be accessed during a request using the function `webapp2.get_request()`.

3.4.9 Registry

A simple dictionary is available in the request object to register instances that are shared during a request: it is the `webapp2.Request.registry` attribute.

A registry dictionary is also available in the *WSGI application object*, to store objects shared across requests.

3.4.10 Learn more about WebOb

WebOb is an open source third-party library. See the WebOb documentation for a detailed API reference and examples.

3.5 Building a Response

The request handler instance builds the response using its response property. This is initialized to an empty WebOb `Response` object by the application.

The response object's acts as a file-like object that can be used for writing the body of the response:

```
class MyHandler(webapp2.RequestHandler):
    def get(self):
        self.response.write("<html><body><p>Hi there!</p></body></html>")
```

The response buffers all output in memory, then sends the final output when the handler exits. webapp2 does not support streaming data to the client.

The `clear()` method erases the contents of the output buffer, leaving it empty.

If the data written to the output stream is a Unicode value, or if the response includes a `Content-Type` header that ends with `; charset=utf-8`, webapp2 encodes the output as UTF-8. By default, the `Content-Type` header is `text/html; charset=utf-8`, including the encoding behavior. If the `Content-Type` is changed to have a different charset, webapp2 assumes the output is a byte string to be sent verbatim.

3.5.1 Setting cookies

Cookies are set in the response object. The methods to handle cookies are:

set_cookie(key, value='', max_age=None, path='/', domain=None, secure=None, httponly=False, version=None, comment=None
Sets a cookie.

delete_cookie(key, path='/', domain=None) Deletes a cookie previously set in the client.

unset_cookie(key) Unsets a cookie previously set in the response object. Note that this doesn't delete the cookie from clients, only from the response.

For example:

```
# Saves a cookie in the client.
response.set_cookie('some_key', 'value', max_age=360, path='/',
                    domain='example.org', secure=True)

# Deletes a cookie previously set in the client.
response.delete_cookie('bad_cookie')

# Cancels a cookie previously set in the response.
response.unset_cookie('some_key')
```

Only the `key` parameter is required. The parameters are:

key Cookie name.

value Cookie value.

max_age Cookie max age in seconds.

path URI path in which the cookie is valid.

domain URI domain in which the cookie is valid.

secure If True, the cookie is only available via HTTPS.

httponly Disallow JavaScript to access the cookie.

version Defines a cookie version number.

comment Defines a cookie comment.

See Also:

How to read cookies from the request object

3.5.2 Common Response attributes

status Status code plus message, e.g., '404 Not Found'. The status can be set passing an `int`, e.g., `request.status = 404`, or including the message, e.g., `request.status = '404 Not Found'`.

status_int Status code as an `int`, e.g., 404.

status_message Status message, e.g., 'Not Found'.

body The contents of the response, as a string.

unicode_body The contents of the response, as a unicode.

headers A dictionary-like object with headers. Keys are case-insensitive. It supports multiple values for a key, but you must use `response.headers.add(key, value)` to add keys. To get all values, use `response.headers.getall(key)`.

headerlist List of headers, as a list of tuples `(header_name, value)`.

charset Character encoding.

content_type 'Content-Type' value from the headers, e.g., `'text/html'`.

content_type_params Dictionary of extra Content-type parameters, e.g., `{'charset': 'utf8'}`.

location 'Location' header variable, used for redirects.

etag 'ETag' header variable. You can automatically generate an etag based on the response body calling `response.md5_etag()`.

3.5.3 Learn more about WebOb

WebOb is an open source third-party library. See the WebOb documentation for a detailed API reference and examples.

3.6 Exception handling

A good app is prepared even when something goes wrong: a service is down, the application didn't expect a given input type or many other errors that can happen in a web application. To react to these cases, we need a good exception handling mechanism and prepare the app to handle the unexpected scenarios.

3.6.1 HTTP exceptions

WebOb provides a collection of exceptions that correspond to HTTP status codes. They all extend a base class, `webob.exc.HTTPException`, also available in webapp2 as `webapp2.HTTPException`.

An `HTTPException` is also a WSGI application, meaning that an instance of it can be returned to be used as response. If an `HTTPException` is not handled, it will be used as a standard response, setting the header status code and a default error message in the body.

3.6.2 Exceptions in handlers

Handlers can catch exceptions implementing the method `webapp2.RequestHandler.handle_exception()`. It is a good idea to define a base class that catches generic exceptions, and if needed override `handle_exception()` in extended classes to set more specific responses.

Here we will define a exception handling function in a base class, and the real app classes extend it:

```
import logging

import webapp2

class BaseHandler(webapp2.RequestHandler):
    def handle_exception(self, exception, debug):
        # Log the error.
        logging.exception(exception)

        # Set a custom message.
        response.write('An error occurred.')

        # If the exception is a HTTPException, use its error code.
        # Otherwise use a generic 500 error code.
        if isinstance(exception, webapp2.HTTPException):
            response.set_status(exception.code)
        else:
            response.set_status(500)

class HomeHandler(BaseHandler):
    def get(self):
        self.response.write('This is the HomeHandler.')
```

```
class ProductListHandler(BaseHandler):
    def get(self):
        self.response.write('This is the ProductListHandler.')
```

If something unexpected happens during the `HomeHandler` or `ProductListHandler` lifetime, `handle_exception()` will catch it because they extend a class that implements exception handling.

You can use exception handling to log errors and display custom messages instead of a generic error. You could also render a template with a friendly message, or return a JSON with an error code, depending on your app.

3.6.3 Exceptions in the WSGI app

Uncaught exceptions can also be handled by the WSGI application. The WSGI app is a good place to handle '404 Not Found' or '500 Internal Server Error' errors, since it serves as a last attempt to handle all uncaught exceptions, including non-registered URI paths or unexpected application behavior.

We catch exceptions in the WSGI app using error handlers registered in `webapp2.WSGIApplication.error_handlers`. This is a dictionary that maps HTTP status codes to callables that will handle the corresponding error code. If the exception is not an `HTTPException`, the status code 500 is used.

Here we set error handlers to handle "404 Not Found" and "500 Internal Server Error":

```
import logging

import webapp2

def handle_404(request, response, exception):
    logging.exception(exception)
    response.write('Oops! I could swear this page was here!')
    response.set_status(404)

def handle_500(request, response, exception):
    logging.exception(exception)
    response.write('A server error occurred!')
    response.set_status(500)

app = webapp2.WSGIApplication([
    webapp2.Route('/', handler='handlers.HomeHandler', name='home')
])
app.error_handlers[404] = handle_404
app.error_handlers[500] = handle_500
```

The error handler can be a simple function that accepts `(request, response, exception)` as parameters, and is responsible for setting the response status code and, if needed, logging the exception.

3.6.4 abort()

The function `webapp2.abort()` is a shortcut to raise one of the HTTP exceptions provided by WebOb: it takes an HTTP status code (403, 404, 500 etc) and raises the corresponding exception.

Use `abort` (or `webapp2.RequestHandler.abort()` inside handlers) to raise an `HTTPException` to be handled by an exception handler. For example, we could call `abort(404)` when a requested item is not found in the database, and have an exception handler ready to handle 404s.

Besides the status code, some extra keyword arguments can be passed to `abort()`:

detail An explanation about the error.

comment An more detailed comment to be included in the response body.

headers Extra response headers to be set.

body_template A string to be used as template for the response body. The default template has the following format, with variables replaced by arguments, if defined:

```
${explanation}<br /><br />
${detail}
${html_comment}
```

3.7 Unit testing

Thanks to WebOb, webapp2 is very testable. Testing a handler is a matter of building a custom `Request` object and calling `get_response()` on it passing the WSGI application.

Let's see an example. First define a simple 'Hello world' handler to be tested:

```
import webapp2

class HelloHandler(webapp2.RequestHandler):
    def get(self):
        self.response.write('Hello, world!')

app = webapp2.WSGIapplication([('/', HelloHandler)])

def main():
    app.run()

if __name__ == '__main__':
    main()
```

To test if this handler returns the correct `'Hello, world!'` response, we build a request object using `Request.blank()` and call `get_response()` on it:

```
import unittest
import webapp2

# from the app main.py
import main

class TestHandlers(unittest.TestCase):
   def test_hello(self):
       # Build a request object passing the URI path to be tested.
       # You can also pass headers, query arguments etc.
       request = webapp2.Request.blank('/')
       # Get a response for that request.
       response = request.get_response(main.app)

       # Let's check if the response is correct.
       self.assertEqual(response.status_int, 200)
       self.assertEqual(response.body, 'Hello, world!')
```

To test different HTTP methods, just change the request object:

```
request = webapp2.Request.blank('/')
request.method = 'POST'
response = request.get_response(main.app)

# Our handler doesn't implement post(), so this response will have a
# status code 405.
self.assertEqual(response.status_int, 405)
```

3.7.1 Request.blank()

`Request.blank(path, environ=None, base_url=None, headers=None, POST=None, **kwargs)` is a class method that creates a new request object for testing purposes. It receives the following parameters:

path A URI path, urlencoded. The path will become path_info, with any query string split off and used.

environ An environ dictionary.

base_url If defined, wsgi.url_scheme, HTTP_HOST and SCRIPT_NAME will be filled in from this value.

headers A list of `(header_name, value)` tuples for the request headers.

POST A dictionary of POST data to be encoded, or a urlencoded string. This is a shortcut to set POST data in the environ. When set, the HTTP method is set to 'POST' and the CONTENT_TYPE is set to 'application/x-www-form-urlencoded'.

kwargs Extra keyword arguments to be passed to `Request.__init__()`.

All necessary keys will be added to the environ, but the values you pass in will take precedence.

3.7.2 app.get_response()

We can also get a response directly from the WSGI application, calling `app.get_response()`. This is a convenience to test the app. It receives the same parameters as `Request.blank()` to build a request and call the application, returning the resulting response:

```
class HelloHandler(webapp2.RequestHandler):
    def get(self):
        self.response.write('Hello, world!')

app = webapp2.WSGIapplication([('/', HelloHandler)])

# Test the app, passing parameters to build a request.
response = app.get_response('/')
assert response.status_int == 200
assert response.body == 'Hello, world!'
```

Testing handlers could not be easier. Check the WebOb documentation for more information about the request and response objects.

3.7.3 Testing App Engine services

If you're using App Engine and need to test an application that uses Datastore, Memcache or other App Engine services, read Local Unit Testing for Python in the official documentation. The App Engine SDK provides the module `google.appengine.ext.testbed` that can be used to setup all the necessary service stubs for testing.

3.8 webapp2_extras

webapp2_extras is a package with common utilities that work well with webapp2. It includes:

- Localization and internationalization support
- Sessions using secure cookies, memcache or datastore
- Extra route classes – to match subdomains and other conveniences
- Support for third party libraries: Jinja2, Mako and Google's ProtoRPC
- Support for threaded environments, so that you can use webapp2 outside of App Engine or in the upcoming App Engine Python 2.7 runtime

Some of these modules (*i18n*, *Jinja2*, *Mako* and *Sessions*) use configuration values that can be set in the WSGI application. When a config key is not set, the modules will use the default values they define.

All configuration keys are optional, except `secret_key` that must be set for *Sessions*. Here is an example that sets the `secret_key` configuration and tests that the session is working:

```
import webapp2
from webapp2_extras import sessions

class BaseHandler(webapp2.RequestHandler):
    def dispatch(self):
        # Get a session store for this request.
        self.session_store = sessions.get_store(request=self.request)

        try:
            # Dispatch the request.
            webapp2.RequestHandler.dispatch(self)
        finally:
            # Save all sessions.
            self.session_store.save_sessions(self.response)

    @webapp2.cached_property
    def session(self):
        # Returns a session using the default cookie key.
        return self.session_store.get_session()

class HomeHandler(BaseHandler):
    def get(self):
        test_value = self.session.get('test-value')
        if test_value:
            self.response.write('Session has this value: %r.' % test_value)
        else:
            self.session['test-value'] = 'Hello, session world!'
            self.response.write('Session is empty.')

config = {}
config['webapp2_extras.sessions'] = {
    'secret_key': 'some-secret-key',
}

app = webapp2.WSGIAppplication([
    ('/', HomeHandler),
], debug=True, config=config)

def main():
```

```
    app.run()

if __name__ == '__main__':
    main()
```

CHAPTER

FOUR

API REFERENCE - WEBAPP2

4.1 webapp2

- WSGI app
  - WSGIApplication
  - RequestContext
- URI routing
  - Router
  - BaseRoute
  - SimpleRoute
  - Route
- Configuration
  - Config
- Request and Response
  - Request
  - Response
- Request handlers
  - RequestHandler
  - RedirectHandler
- Utilities
  - cached_property
  - get_app()
  - get_request()
  - redirect()
  - redirect_to()
  - uri_for()
  - abort()
  - import_string()

- urlunsplit()

4.1.1 WSGI app

See Also:

The WSGI application

class webapp2.**WSGIApplication**(*routes=None*, *debug=False*, *config=None*)
A WSGI-compliant application.

request_class
Class used for the request object.

response_class
Class used for the response object.

request_context_class
Class used for the request context object.

router_class
Class used for the router object.

config_class
Class used for the configuration object.

debug
A general purpose flag to indicate development mode: if True, uncaught exceptions are raised instead of using HTTPInternalServerError.

router
A Router instance with all URIs registered for the application.

config
A Config instance with the application configuration.

registry
A dictionary to register objects used during the app lifetime.

error_handlers
A dictionary mapping HTTP error codes to callables to handle those HTTP exceptions. See handle_exception().

app
Active WSGIApplication instance. See set_globals().

request
Active Request instance. See set_globals().

active_instance
Same as app, for webapp compatibility. See set_globals().

allowed_methods
Allowed request methods.

__init__(*routes=None*, *debug=False*, *config=None*)
Initializes the WSGI application.

Parameters

- **routes** – A sequence of Route instances or, for simple routes, tuples (regex, handler).

- **debug** – True to enable debug mode, False otherwise.
- **config** – A configuration dictionary for the application.

__call__(*environ*, *start_response*)
Called by WSGI when a request comes in.

Parameters

- **environ** – A WSGI environment.
- **start_response** – A callable accepting a status code, a list of headers and an optional exception context to start the response.

Returns An iterable with the response to return to the client.

set_globals(*app=None*, *request=None*)
Registers the global variables for app and request.

If `webapp2_extras.local` is available the app and request class attributes are assigned to a proxy object that returns them using thread-local, making the application thread-safe. This can also be used in environments that don't support threading.

If `webapp2_extras.local` is not available app and request will be assigned directly as class attributes. This should only be used in non-threaded environments (e.g., App Engine Python 2.5).

Parameters

- **app** – A `WSGIApplication` instance.
- **request** – A `Request` instance.

clear_globals()
Clears global variables. See `set_globals()`.

handle_exception(*request*, *response*, *e*)
Handles a uncaught exception occurred in `__call__()`.

Uncaught exceptions can be handled by error handlers registered in `error_handlers`. This is a dictionary that maps HTTP status codes to callables that will handle the corresponding error code. If the exception is not an `HTTPException`, the status code 500 is used.

The error handlers receive (request, response, exception) and can be a callable or a string in dotted notation to be lazily imported.

If no error handler is found, the exception is re-raised.

Based on idea from Flask.

Parameters

- **request** – A `Request` instance.
- **request** – A `Response` instance.
- **e** – The uncaught exception.

Returns The returned value from the error handler.

run(*bare=False*)
Runs this WSGI-compliant application in a CGI environment.

This uses functions provided by `google.appengine.ext.webapp.util`, if available: `run_bare_wsgi_app` and `run_wsgi_app`.

Otherwise, it uses `wsgiref.handlers.CGIHandler().run()`.

Parameters **bare** – If True, doesn't add registered WSGI middleware: use `run_bare_wsgi_app` instead of `run_wsgi_app`.

get_response(**args*, ***kwargs*)
Creates a request and returns a response for this app.

This is a convenience for unit testing purposes. It receives parameters to build a request and calls the application, returning the resulting response:

```
class HelloHandler(webapp2.RequestHandler):
    def get(self):
        self.response.write('Hello, world!')

app = webapp2.WSGIapplication([('/', HelloHandler)])

# Test the app, passing parameters to build a request.
response = app.get_response('/')
assert response.status_int == 200
assert response.body == 'Hello, world!'
```

Parameters

- **args** – Positional arguments to be passed to `Request.blank()`.
- **kwargs** – Keyword arguments to be passed to `Request.blank()`.

Returns A `Response` object.

class webapp2.**RequestContext**(*app*, *environ*)
Context for a single request.

The context is responsible for setting and cleaning global variables for a request.

__init__(*app*, *environ*)
Initializes the request context.

Parameters

- **app** – An `WSGIApplication` instance.
- **environ** – A WSGI environment dictionary.

__enter__()
Enters the request context.

Returns A tuple `(request, response)`.

__exit__(*exc_type*, *exc_value*, *traceback*)
Exits the request context.

This release the context locals except if an exception is caught in debug mode. In this case they are kept to be inspected.

4.1.2 URI routing

See Also:

Router and *URI routing*

class webapp2.**Router**(*routes=None*)
A URI router used to match, dispatch and build URIs.

route_class
Class used when the route is set as a tuple.

__init__(*routes=None*)
Initializes the router.

Parameters **routes** – A sequence of `Route` instances or, for simple routes, tuples `(regex, handler)`.

add(*route*)
Adds a route to this router.

Parameters **route** – A `Route` instance or, for simple routes, a tuple `(regex, handler)`.

match(*request*)
Matches all routes against a request object.

The first one that matches is returned.

Parameters **request** – A `Request` instance.

Returns A tuple `(route, args, kwargs)` if a route matched, or None.

Raises `exc.HTTPNotFound` if no route matched or `exc.HTTPMethodNotAllowed` if a route matched but the HTTP method was not allowed.

build(*request*, *name*, *args*, *kwargs*)
Returns a URI for a named `Route`.

Parameters

- **request** – The current `Request` object.
- **name** – The route name.
- **args** – Tuple of positional arguments to build the URI. All positional variables defined in the route must be passed and must conform to the format set in the route. Extra arguments are ignored.
- **kwargs** – Dictionary of keyword arguments to build the URI. All variables not set in the route default values must be passed and must conform to the format set in the route. Extra keywords are appended as a query string.

  A few keywords have special meaning:

  - **_full**: If True, builds an absolute URI.
  - **_scheme**: URI scheme, e.g., *http* or *https*. If defined, an absolute URI is always returned.
  - **_netloc**: Network location, e.g., *www.google.com*. If defined, an absolute URI is always returned.
  - **_fragment**: If set, appends a fragment (or "anchor") to the generated URI.

Returns An absolute or relative URI.

dispatch(*request*, *response*)
Dispatches a handler.

Parameters

- **request** – A `Request` instance.
- **response** – A `Response` instance.

Raises `exc.HTTPNotFound` if no route matched or `exc.HTTPMethodNotAllowed` if a route matched but the HTTP method was not allowed.

Returns The returned value from the handler.

adapt(*handler*)
Adapts a handler for dispatching.

Because handlers use or implement different dispatching mechanisms, they can be wrapped to use a unified API for dispatching. This way webapp2 can support, for example, a `RequestHandler` class and function views or, for compatibility purposes, a `webapp.RequestHandler` class. The adapters follow the same router dispatching API but dispatch each handler type differently.

Parameters **handler** – A handler callable.

Returns A wrapped handler callable.

default_matcher(*request*)
Matches all routes against a request object.

The first one that matches is returned.

Parameters **request** – A `Request` instance.

Returns A tuple `(route, args, kwargs)` if a route matched, or None.

Raises `exc.HTTPNotFound` if no route matched or `exc.HTTPMethodNotAllowed` if a route matched but the HTTP method was not allowed.

default_builder(*request*, *name*, *args*, *kwargs*)
Returns a URI for a named `Route`.

Parameters

- **request** – The current `Request` object.
- **name** – The route name.
- **args** – Tuple of positional arguments to build the URI. All positional variables defined in the route must be passed and must conform to the format set in the route. Extra arguments are ignored.
- **kwargs** – Dictionary of keyword arguments to build the URI. All variables not set in the route default values must be passed and must conform to the format set in the route. Extra keywords are appended as a query string.

  A few keywords have special meaning:

  - **_full**: If True, builds an absolute URI.
  - **_scheme**: URI scheme, e.g., *http* or *https*. If defined, an absolute URI is always returned.
  - **_netloc**: Network location, e.g., *www.google.com*. If defined, an absolute URI is always returned.
  - **_fragment**: If set, appends a fragment (or "anchor") to the generated URI.

Returns An absolute or relative URI.

default_dispatcher(*request*, *response*)
Dispatches a handler.

Parameters

- **request** – A `Request` instance.

- **response** – A `Response` instance.

Raises `exc.HTTPNotFound` if no route matched or `exc.HTTPMethodNotAllowed` if a route matched but the HTTP method was not allowed.

Returns The returned value from the handler.

default_adapter(*handler*)
Adapts a handler for dispatching.

Because handlers use or implement different dispatching mechanisms, they can be wrapped to use a unified API for dispatching. This way webapp2 can support, for example, a `RequestHandler` class and function views or, for compatibility purposes, a `webapp.RequestHandler` class. The adapters follow the same router dispatching API but dispatch each handler type differently.

Parameters **handler** – A handler callable.

Returns A wrapped handler callable.

set_matcher(*func*)
Sets the function called to match URIs.

Parameters **func** – A function that receives `(router, request)` and returns a tuple `(route, args, kwargs)` if any route matches, or raise `exc.HTTPNotFound` if no route matched or `exc.HTTPMethodNotAllowed` if a route matched but the HTTP method was not allowed.

set_builder(*func*)
Sets the function called to build URIs.

Parameters **func** – A function that receives `(router, request, name, args, kwargs)` and returns a URI.

set_dispatcher(*func*)
Sets the function called to dispatch the handler.

Parameters **func** – A function that receives `(router, request, response)` and returns the value returned by the dispatched handler.

set_adapter(*func*)
Sets the function that adapts loaded handlers for dispatching.

Parameters **func** – A function that receives `(router, handler)` and returns a handler callable.

class webapp2.**BaseRoute**(*template*, *handler=None*, *name=None*, *build_only=False*)
Interface for URI routes.

template
The regex template.

name
Route name, used to build URIs.

handler
The handler or string in dotted notation to be lazily imported.

handler_method
The custom handler method, if handler is a class.

handler_adapter
The handler, imported and ready for dispatching.

build_only
True if this route is only used for URI generation and never matches.

match(*request*)
Matches all routes against a request object.

The first one that matches is returned.

Parameters **request** – A `Request` instance.

Returns A tuple `(route, args, kwargs)` if a route matched, or None.

build(*request*, *args*, *kwargs*)
Returns a URI for this route.

Parameters

- **request** – The current `Request` object.
- **args** – Tuple of positional arguments to build the URI.
- **kwargs** – Dictionary of keyword arguments to build the URI.

Returns An absolute or relative URI.

get_routes()
Generator to get all routes from a route.

Yields This route or all nested routes that it contains.

get_match_routes()
Generator to get all routes that can be matched from a route.

Match routes must implement `match()`.

Yields This route or all nested routes that can be matched.

get_build_routes()
Generator to get all routes that can be built from a route.

Build routes must implement `build()`.

Yields A tuple `(name, route)` for all nested routes that can be built.

class webapp2.**SimpleRoute**(*template*, *handler=None*, *name=None*, *build_only=False*)
A route that is compatible with webapp's routing mechanism.

URI building is not implemented as webapp has rudimentar support for it, and this is the most unknown webapp feature anyway.

__init__(*template*, *handler=None*, *name=None*, *build_only=False*)
Initializes this route.

Parameters

- **template** – A regex to be matched.
- **handler** – A callable or string in dotted notation to be lazily imported, e.g., `'my.module.MyHandler'` or `'my.module.my_function'`.
- **name** – The name of this route, used to build URIs based on it.
- **build_only** – If True, this route never matches and is used only to build URIs.

match(*request*)
Matches this route against the current request.

See Also:

BaseRoute.match().

class webapp2.**Route**(*template*, *handler=None*, *name=None*, *defaults=None*, *build_only=False*, *handler_method=None*, *methods=None*, *schemes=None*)

A route definition that maps a URI path to a handler.

The initial concept was based on Another Do-It-Yourself Framework, by Ian Bicking.

__init__(*template*, *handler=None*, *name=None*, *defaults=None*, *build_only=False*, *handler_method=None*, *methods=None*, *schemes=None*)

Initializes this route.

Parameters

- **template** – A route template to match against the request path. A template can have variables enclosed by <> that define a name, a regular expression or both. Examples:

Format	Example
<name>	'/blog/<year>/<month>'
<:regex>	'/blog/<:\d{4}>/<:\d{2}>'
<name:regex>	'/blog/<year:\d{4}>/<month:\d{2}>'

  The same template can mix parts with name, regular expression or both.

  If the name is set, the value of the matched regular expression is passed as keyword argument to the handler. Otherwise it is passed as positional argument.

  If only the name is set, it will match anything except a slash. So these routes are equivalent:

```
Route('/<user_id>/settings', handler=SettingsHandler, name='user-settings')
Route('/<user_id:[^/]+>/settings', handler=SettingsHandler, name='user-settings')
```

  Note: The handler only receives `*args` if no named variables are set. Otherwise, the handler only receives `**kwargs`. This allows you to set regular expressions that are not captured: just mix named and unnamed variables and the handler will only receive the named ones.

- **handler** – A callable or string in dotted notation to be lazily imported, e.g., `'my.module.MyHandler'` or `'my.module.my_function'`. It is possible to define a method if the callable is a class, separating it by a colon: `'my.module.MyHandler:my_method'`. This is a shortcut and has the same effect as defining the *handler_method* parameter.
- **name** – The name of this route, used to build URIs based on it.
- **defaults** – Default or extra keywords to be returned by this route. Values also present in the route variables are used to build the URI when they are missing.
- **build_only** – If True, this route never matches and is used only to build URIs.
- **handler_method** – The name of a custom handler method to be called, in case *handler* is a class. If not defined, the default behavior is to call the handler method correspondent to the HTTP request method in lower case (e.g., *get()*, *post()* etc).
- **methods** – A sequence of HTTP methods. If set, the route will only match if the request method is allowed.
- **schemes** – A sequence of URI schemes, e.g., `['http']` or `['https']`. If set, the route will only match requests with these schemes.

match(*request*)
Matches this route against the current request.

Raises `exc.HTTPMethodNotAllowed` if the route defines `methods` and the request method isn't allowed.

See Also:

`BaseRoute.match()`.

build(*request*, *args*, *kwargs*)
Returns a URI for this route.

See Also:

`Router.build()`.

4.1.3 Configuration

See Also:

Config

class webapp2.**Config**(*defaults=None*)
A simple configuration dictionary for the `WSGIApplication`.

load_config(*key*, *default_values=None*, *user_values=None*, *required_keys=None*)
Returns a configuration for a given key.

This can be used by objects that define a default configuration. It will update the app configuration with the default values the first time it is requested, and mark the key as loaded.

Parameters

- **key** – A configuration key.
- **default_values** – Default values defined by a module or class.
- **user_values** – User values, used when an object can be initialized with configuration. This overrides the app configuration.
- **required_keys** – Keys that can not be None.

Raises Exception, when a required key is not set or is None.

4.1.4 Request and Response

See Also:

Request data and *Building a Response*

class webapp2.**Request**(*environ*, **args*, ***kwargs*)
Abstraction for an HTTP request.

Most extra methods and attributes are ported from webapp. Check the WebOb documentation for the ones not listed here.

app
A reference to the active `WSGIApplication` instance.

response
A reference to the active `Response` instance.

route
A reference to the matched Route.

route_args
The matched route positional arguments.

route_kwargs
The matched route keyword arguments.

registry
A dictionary to register objects used during the request lifetime.

__init__(*environ*, **args*, ***kwargs*)
Constructs a Request object from a WSGI environment.

Parameters **environ** – A WSGI-compliant environment dictionary.

get(*argument_name*, *default_value=''*, *allow_multiple=False*)
Returns the query or POST argument with the given name.

We parse the query string and POST payload lazily, so this will be a slower operation on the first call.

Parameters

- **argument_name** – The name of the query or POST argument.
- **default_value** – The value to return if the given argument is not present.
- **allow_multiple** – Return a list of values with the given name (deprecated).

Returns If allow_multiple is False (which it is by default), we return the first value with the given name given in the request. If it is True, we always return a list.

get_all(*argument_name*, *default_value=None*)
Returns a list of query or POST arguments with the given name.

We parse the query string and POST payload lazily, so this will be a slower operation on the first call.

Parameters

- **argument_name** – The name of the query or POST argument.
- **default_value** – The value to return if the given argument is not present, None may not be used as a default, if it is then an empty list will be returned instead.

Returns A (possibly empty) list of values.

arguments()
Returns a list of the arguments provided in the query and/or POST.

The return value is a list of strings.

get_range(*name*, *min_value=None*, *max_value=None*, *default=0*)
Parses the given int argument, limiting it to the given range.

Parameters

- **name** – The name of the argument.
- **min_value** – The minimum int value of the argument (if any).
- **max_value** – The maximum int value of the argument (if any).
- **default** – The default value of the argument if it is not given.

Returns An int within the given range for the argument.

class webapp2.**Response**(**args*, ***kwargs*)
Abstraction for an HTTP response.

Most extra methods and attributes are ported from webapp. Check the WebOb documentation for the ones not listed here.

Differences from webapp.Response:

- `out` is not a `StringIO.StringIO` instance. Instead it is the response itself, as it has the method `write()`.
- As in WebOb, `status` is the code plus message, e.g., '200 OK', while in webapp it is the integer code. The status code as an integer is available in `status_int`, and the status message is available in `status_message`.
- `response.headers` raises an exception when a key that doesn't exist is accessed or deleted, differently from `wsgiref.headers.Headers`.

__init__(**args*, ***kwargs*)
Constructs a response with the default settings.

status
The status string, including code and message.

status_message
The response status message, as a string.

has_error()
Indicates whether the response was an error response.

clear()
Clears all data written to the output stream so that it is empty.

wsgi_write(*start_response*)
Writes this response using using the given WSGI function.

This is only here for compatibility with `webapp.WSGIApplication`.

Parameters **start_response** – The WSGI-compatible start_response function.

static **http_status_message**(*code*)
Returns the default HTTP status message for the given code.

Parameters **code** – The HTTP code for which we want a message.

4.1.5 Request handlers

See Also:

Request handlers

class webapp2.**RequestHandler**(*request=None*, *response=None*)
Base HTTP request handler.

Implements most of `webapp.RequestHandler` interface.

app
A `WSGIApplication` instance.

request
A `Request` instance.

response
A `Response` instance.

__init__(*request=None*, *response=None*)
Initializes this request handler with the given WSGI application, Request and Response.

When instantiated by `webapp.WSGIApplication`, request and response are not set on instantiation. Instead, initialize() is called right after the handler is created to set them.

Also in webapp dispatching is done by the WSGI app, while webapp2 does it here to allow more flexibility in extended classes: handlers can wrap `dispatch()` to check for conditions before executing the requested method and/or post-process the response.

Note: Parameters are optional only to support webapp's constructor which doesn't take any arguments. Consider them as required.

Parameters

- **request** – A `Request` instance.
- **response** – A `Response` instance.

initialize(*request*, *response*)
Initializes this request handler with the given WSGI application, Request and Response.

Parameters

- **request** – A `Request` instance.
- **response** – A `Response` instance.

dispatch()
Dispatches the request.

This will first check if there's a handler_method defined in the matched route, and if not it'll use the method correspondent to the request method (`get()`, `post()` etc).

error(*code*)
Clears the response and sets the given HTTP status code.

This doesn't stop code execution; for this, use `abort()`.

Parameters **code** – HTTP status error code (e.g., 501).

abort(*code*, **args*, ***kwargs*)
Raises an `HTTPException`.

This stops code execution, leaving the HTTP exception to be handled by an exception handler.

Parameters

- **code** – HTTP status code (e.g., 404).
- **args** – Positional arguments to be passed to the exception class.
- **kwargs** – Keyword arguments to be passed to the exception class.

redirect(*uri*, *permanent=False*, *abort=False*, *code=None*, *body=None*)
Issues an HTTP redirect to the given relative URI.

The arguments are described in `redirect()`.

redirect_to(*_name*, *_permanent=False*, *_abort=False*, *_code=None*, *_body=None*, **args*, ***kwargs*)
Convenience method mixing `redirect()` and `uri_for()`.

The arguments are described in `redirect()` and `uri_for()`.

uri_for(*_name*, **args*, ***kwargs*)
Returns a URI for a named `Route`.

See Also:

`Router.build()`.

handle_exception(*exception*, *debug*)
Called if this handler throws an exception during execution.

The default behavior is to re-raise the exception to be handled by `WSGIApplication.handle_exception()`.

Parameters

- **exception** – The exception that was thrown.
- **debug_mode** – True if the web application is running in debug mode.

class webapp2.**RedirectHandler**(*request=None*, *response=None*)
Redirects to the given URI for all GET requests.

This is intended to be used when defining URI routes. You must provide at least the keyword argument *url* in the route default values. Example:

```
def get_redirect_url(handler, *args, **kwargs):
    return handler.uri_for('new-route-name')

app = WSGIApplication([
    Route('/old-url', RedirectHandler, defaults={'_uri': '/new-url'}),
    Route('/other-old-url', RedirectHandler, defaults={'_uri': get_redirect_url}),
])
```

Based on idea from Tornado.

get(**args*, ***kwargs*)
Performs a redirect.

Two keyword arguments can be passed through the URI route:

- **_uri**: A URI string or a callable that returns a URI. The callable is called passing `(handler, *args, **kwargs)` as arguments.
- **_code**: The redirect status code. Default is 301 (permanent redirect).

4.1.6 Utilities

These are some other utilities also available for general use.

class webapp2.**cached_property**(*func*, *name=None*, *doc=None*)
A decorator that converts a function into a lazy property.

The function wrapped is called the first time to retrieve the result and then that calculated result is used the next time you access the value:

```
class Foo(object):

    @cached_property
    def foo(self):
        # calculate something important here
        return 42
```

The class has to have a *__dict__* in order for this property to work.

Note: Implementation detail: this property is implemented as non-data descriptor. non-data descriptors are only invoked if there is no entry with the same name in the instance's __dict__. this allows us to completely get rid of the access function call overhead. If one choses to invoke __get__ by hand the property will still work as expected because the lookup logic is replicated in __get__ for manual invocation.

This class was ported from Werkzeug and Flask.

webapp2.**get_app**()
Returns the active app instance.

Returns A `WSGIApplication` instance.

webapp2.**get_request**()
Returns the active request instance.

Returns A `Request` instance.

webapp2.**redirect**(*uri*, *permanent=False*, *abort=False*, *code=None*, *body=None*, *request=None*, *response=None*)
Issues an HTTP redirect to the given relative URI.

This won't stop code execution unless **abort** is True. A common practice is to return when calling this method:

```
return redirect('/some-path')
```

Parameters

- **uri** – A relative or absolute URI (e.g., `'../flowers.html'`).
- **permanent** – If True, uses a 301 redirect instead of a 302 redirect.
- **abort** – If True, raises an exception to perform the redirect.
- **code** – The redirect status code. Supported codes are 301, 302, 303, 305, and 307. 300 is not supported because it's not a real redirect and 304 because it's the answer for a request with defined `If-Modified-Since` headers.
- **body** – Response body, if any.
- **request** – Optional request object. If not set, uses `get_request()`.
- **response** – Optional response object. If not set, a new response is created.

Returns A `Response` instance.

webapp2.**redirect_to**(*_name*, *_permanent=False*, *_abort=False*, *_code=None*, *_body=None*, *_request=None*, *_response=None*, **args*, ***kwargs*)
Convenience function mixing `redirect()` and `uri_for()`.

Issues an HTTP redirect to a named URI built using `uri_for()`.

Parameters

- **_name** – The route name to redirect to.
- **args** – Positional arguments to build the URI.
- **kwargs** – Keyword arguments to build the URI.

Returns A `Response` instance.

The other arguments are described in `redirect()`.

webapp2.**uri_for**(*_name*, *_request=None*, **args*, ***kwargs*)
A standalone uri_for version that can be passed to templates.

See Also:

Router.build().

webapp2.**abort**(*code*, **args*, ***kwargs*)
Raises an HTTPException.

Parameters

- **code** – An integer that represents a valid HTTP status code.
- **args** – Positional arguments to instantiate the exception.
- **kwargs** – Keyword arguments to instantiate the exception.

webapp2.**import_string**(*import_name*, *silent=False*)
Imports an object based on a string in dotted notation.

Simplified version of the function with same name from Werkzeug.

Parameters

- **import_name** – String in dotted notation of the object to be imported.
- **silent** – If True, import or attribute errors are ignored and None is returned instead of raising an exception.

Returns The imported object.

CHAPTER

FIVE

API REFERENCE - WEBAPP2_EXTRAS

5.1 Auth

Warning: This is an experimental module. The API is subject to changes.

webapp2_extras.auth.**default_config**
Default configuration values for this module. Keys are:

user_model User model which authenticates custom users and tokens. Can also be a string in dotted notation to be lazily imported. Default is `webapp2_extras.appengine.auth.models.User`.

session_backend Name of the session backend to be used. Default is *securecookie*.

cookie_name Name of the cookie to save the auth session. Default is *auth*.

token_max_age Number of seconds of inactivity after which an auth token is invalidated. The same value is used to set the `max_age` for persistent auth sessions. Default is 86400 * 7 * 3 (3 weeks).

token_new_age Number of seconds after which a new token is written to the database. Use this to limit database writes; set to None to write on all requests. Default is 86400 (1 day).

token_cache_age Number of seconds after which a token must be checked in the database. Use this to limit database reads; set to None to read on all requests. Default is 3600 (1 hour).

user_attributes A list of extra user attributes to be stored in the session. Default is an empty list.

class webapp2_extras.auth.**AuthStore**(*app*, *config=None*)
Provides common utilities and configuration for `Auth`.

__init__(*app*, *config=None*)
Initializes the session store.

Parameters

- **app** – A `webapp2.WSGIApplication` instance.
- **config** – A dictionary of configuration values to be overridden. See the available keys in `default_config`.

class webapp2_extras.auth.**Auth**(*request*)
Authentication provider for a single request.

__init__(*request*)
Initializes the auth provider for a request.

Parameters **request** – A `webapp2.Request` instance.

get_user_by_session(*save_session=True*)
Returns a user based on the current session.

Parameters **save_session** – If True, saves the user in the session if authentication succeeds.

Returns A user dict or None.

get_user_by_token(*auth_id*, *token*, *token_ts=None*, *cache=None*, *cache_ts=None*, *remember=False*, *save_session=True*)
Returns a user based on an authentication token.

Parameters

- **auth_id** – Authentication id.
- **token** – Authentication token.
- **token_ts** – Token timestamp, used to perform pre-validation.
- **cache** – Cached user data (from the session).
- **cache_ts** – Cache timestamp.
- **remember** – If True, saves permanent sessions.
- **save_session** – If True, saves the user in the session if authentication succeeds.

Returns A user dict or None.

get_user_by_password(*auth_id*, *password*, *remember=False*, *save_session=True*, *silent=False*)
Returns a user based on password credentials.

Parameters

- **auth_id** – Authentication id.
- **password** – User password.
- **remember** – If True, saves permanent sessions.
- **save_session** – If True, saves the user in the session if authentication succeeds.
- **silent** – If True, raises an exception if auth_id or password are invalid.

Returns A user dict or None.

Raises `InvalidAuthIdError` or `InvalidPasswordError`.

set_session(*user*, *token=None*, *token_ts=None*, *cache_ts=None*, *remember=False*, ***session_args*)
Saves a user in the session.

Parameters

- **user** – A dictionary with user data.
- **token** – A unique token to be persisted. If None, a new one is created.
- **token_ts** – Token timestamp. If None, a new one is created.
- **cache_ts** – Token cache timestamp. If None, a new one is created.
- **session_args** – Keyword arguments to set the session arguments.

Remember If True, session is set to be persisted.

unset_session()
Removes a user from the session and invalidates the auth token.

webapp2_extras.auth.**get_store**(*factory=<class 'webapp2_extras.auth.AuthStore'>*, *key='webapp2_extras.auth.Auth'*, *app=None*)

Returns an instance of `AuthStore` from the app registry.

It'll try to get it from the current app registry, and if it is not registered it'll be instantiated and registered. A second call to this function will return the same instance.

Parameters

- **factory** – The callable used to build and register the instance if it is not yet registered. The default is the class `AuthStore` itself.
- **key** – The key used to store the instance in the registry. A default is used if it is not set.
- **app** – A `webapp2.WSGIApplication` instance used to store the instance. The active app is used if it is not set.

webapp2_extras.auth.**set_store**(*store*, *key='webapp2_extras.auth.Auth'*, *app=None*)

Sets an instance of `AuthStore` in the app registry.

Parameters

- **store** – An instance of `AuthStore`.
- **key** – The key used to retrieve the instance from the registry. A default is used if it is not set.
- **request** – A `webapp2.WSGIApplication` instance used to retrieve the instance. The active app is used if it is not set.

webapp2_extras.auth.**get_auth**(*factory=<class 'webapp2_extras.auth.Auth'>*, *key='webapp2_extras.auth.Auth'*, *request=None*)

Returns an instance of `Auth` from the request registry.

It'll try to get it from the current request registry, and if it is not registered it'll be instantiated and registered. A second call to this function will return the same instance.

Parameters

- **factory** – The callable used to build and register the instance if it is not yet registered. The default is the class `Auth` itself.
- **key** – The key used to store the instance in the registry. A default is used if it is not set.
- **request** – A `webapp2.Request` instance used to store the instance. The active request is used if it is not set.

webapp2_extras.auth.**set_auth**(*auth*, *key='webapp2_extras.auth.Auth'*, *request=None*)

Sets an instance of `Auth` in the request registry.

Parameters

- **auth** – An instance of `Auth`.
- **key** – The key used to retrieve the instance from the registry. A default is used if it is not set.
- **request** – A `webapp2.Request` instance used to retrieve the instance. The active request is used if it is not set.

5.2 i18n

This module provides internationalization and localization support for webapp2.

To use it, you must add the `babel` and `pytz` packages to your application directory (for App Engine) or install it in your virtual environment (for other servers).

You can download `babel` and `pytz` from the following locations:

http://babel.edgewall.org/ http://pypi.python.org/pypi/gaepytz

5.3 Jinja2

This module provides Jinja2 template support for webapp2.

To use it, you must add the `jinja2` package to your application directory (for App Engine) or install it in your virtual environment (for other servers).

You can download `jinja2` from PyPi:

http://pypi.python.org/pypi/Jinja2

Learn more about Jinja2:

http://jinja.pocoo.org/

`webapp2_extras.jinja2.`**`default_config`**
Default configuration values for this module. Keys are:

template_path Directory for templates. Default is *templates*.

compiled_path Target for compiled templates. If set, uses the loader for compiled templates in production. If it ends with a '.zip' it will be treated as a zip file. Default is None.

force_compiled Forces the use of compiled templates even in the development server.

environment_args Keyword arguments used to instantiate the Jinja2 environment. By default autoescaping is enabled and two extensions are set: `jinja2.ext.autoescape` and `jinja2.ext.with_`. For production it may be a good idea to set 'auto_reload' to False – we don't need to check if templates changed after deployed.

globals Extra global variables for the Jinja2 environment.

filters Extra filters for the Jinja2 environment.

class `webapp2_extras.jinja2.`**`Jinja2`**(*app*, *config=None*)
Wrapper for configurable and cached Jinja2 environment.

To used it, set it as a cached property in a base *RequestHandler*:

```
import webapp2

from webapp2_extras import jinja2

class BaseHandler(webapp2.RequestHandler):

    @webapp2.cached_property
    def jinja2(self):
        # Returns a Jinja2 renderer cached in the app registry.
        return jinja2.get_jinja2(app=self.app)

    def render_response(self, _template, **context):
        # Renders a template and writes the result to the response.
        rv = self.jinja2.render_template(_template, **context)
        self.response.write(rv)
```

Then extended handlers can render templates directly:

```
class MyHandler(BaseHandler):
    def get(self):
        context = {'message': 'Hello, world!'}
        self.render_response('my_template.html', **context)
```

__init__(*app*, *config=None*)
Initializes the Jinja2 object.

Parameters

- **app** – A `webapp2.WSGIApplication` instance.
- **config** – A dictionary of configuration values to be overridden. See the available keys in `default_config`.

render_template(*_filename*, ***context*)
Renders a template and returns a response object.

Parameters

- **_filename** – The template filename, related to the templates directory.
- **context** – Keyword arguments used as variables in the rendered template. These will override values set in the request context.

Returns A rendered template.

get_template_attribute(*filename*, *attribute*)
Loads a macro (or variable) a template exports. This can be used to invoke a macro from within Python code. If you for example have a template named *_foo.html* with the following contents:

```
{% macro hello(name) %}Hello {{ name }}!{% endmacro %}
```

You can access this from Python code like this:

```
hello = get_template_attribute('_foo.html', 'hello')
return hello('World')
```

This function comes from *Flask*.

Parameters

- **filename** – The template filename.
- **attribute** – The name of the variable of macro to acccess.

webapp2_extras.jinja2.**get_jinja2**(*factory=<class 'webapp2_extras.jinja2.Jinja2'>*, *key='webapp2_extras.jinja2.Jinja2'*, *app=None*)
Returns an instance of `Jinja2` from the app registry.

It'll try to get it from the current app registry, and if it is not registered it'll be instantiated and registered. A second call to this function will return the same instance.

Parameters

- **factory** – The callable used to build and register the instance if it is not yet registered. The default is the class `Jinja2` itself.
- **key** – The key used to store the instance in the registry. A default is used if it is not set.
- **app** – A `webapp2.WSGIApplication` instance used to store the instance. The active app is used if it is not set.

webapp2_extras.jinja2.**set_jinja2**(*jinja2*, *key='webapp2_extras.jinja2.Jinja2'*, *app=None*)
Sets an instance of Jinja2 in the app registry.

Parameters

- **store** – An instance of Jinja2.
- **key** – The key used to retrieve the instance from the registry. A default is used if it is not set.
- **request** – A webapp2.WSGIApplication instance used to retrieve the instance. The active app is used if it is not set.

5.4 JSON

This is a wrapper for the json module: it will use simplejson if available, or the json module from Python >= 2.6 if available, and as a last resource the django.utils.simplejson module on App Engine.

It will also escape forward slashes and, by default, output the serialized JSON in a compact format, eliminating white spaces.

Some convenience functions are also available to encode and decode to and from base64 and to quote or unquote the values.

webapp2_extras.json.**encode**(*value*, **args*, ***kwargs*)
Serializes a value to JSON.

This comes from Tornado.

Parameters

- **value** – A value to be serialized.
- **args** – Extra arguments to be passed to *json.dumps()*.
- **kwargs** – Extra keyword arguments to be passed to *json.dumps()*.

Returns The serialized value.

webapp2_extras.json.**decode**(*value*, **args*, ***kwargs*)
Deserializes a value from JSON.

This comes from Tornado.

Parameters

- **value** – A value to be deserialized.
- **args** – Extra arguments to be passed to *json.loads()*.
- **kwargs** – Extra keyword arguments to be passed to *json.loads()*.

Returns The deserialized value.

webapp2_extras.json.**b64encode**(*value*, **args*, ***kwargs*)
Serializes a value to JSON and encodes it using base64.

Parameters and return value are the same from encode().

webapp2_extras.json.**b64decode**(*value*, **args*, ***kwargs*)
Decodes a value using base64 and deserializes it from JSON.

Parameters and return value are the same from decode().

webapp2_extras.json.**quote**(*value*, **args*, ***kwargs*)
Serializes a value to JSON and encodes it using urllib.quote.

Parameters and return value are the same from `encode()`.

webapp2_extras.json.**unquote**(*value*, **args*, ***kwargs*)
Decodes a value using urllib.unquote and deserializes it from JSON.

Parameters and return value are the same from `decode()`.

5.5 Local

This module implements thread-local utilities.

class webapp2_extras.local.**Local**
A container for thread-local objects.

Attributes are assigned or retrieved using the current thread.

class webapp2_extras.local.**LocalProxy**(*local*, *name=None*)
Acts as a proxy for a local object.

Forwards all operations to a proxied object. The only operations not supported for forwarding are right handed operands and any kind of assignment.

Example usage:

```
from webapp2_extras import Local
l = Local()

# these are proxies
request = l('request')
user = l('user')
```

Whenever something is bound to l.user or l.request the proxy objects will forward all operations. If no object is bound a `RuntimeError` will be raised.

To create proxies to `Local` object, call the object as shown above. If you want to have a proxy to an object looked up by a function, you can pass a function to the `LocalProxy` constructor:

```
route_kwargs = LocalProxy(lambda: webapp2.get_request().route_kwargs)
```

5.6 Mako

This module provides Mako template support for webapp2.

To use it, you must add the `mako` package to your application directory (for App Engine) or install it in your virtual environment (for other servers).

You can download `mako` from PyPi:

http://pypi.python.org/pypi/Mako

Learn more about Mako:

http://www.makotemplates.org/

webapp2_extras.mako.**default_config**
Default configuration values for this module. Keys are:

template_path Directory for templates. Default is *templates*.

class webapp2_extras.mako.**Mako**(*app*, *config=None*)
Wrapper for configurable and cached Mako environment.

To used it, set it as a cached property in a base *RequestHandler*:

```
import webapp2

from webapp2_extras import mako

class BaseHandler(webapp2.RequestHandler):

    @webapp2.cached_property
    def mako(self):
        # Returns a Mako renderer cached in the app registry.
        return mako.get_mako(app=self.app)

    def render_response(self, _template, **context):
        # Renders a template and writes the result to the response.
        rv = self.mako.render_template(_template, **context)
        self.response.write(rv)
```

Then extended handlers can render templates directly:

```
class MyHandler(BaseHandler):
    def get(self):
        context = {'message': 'Hello, world!'}
        self.render_response('my_template.html', **context)
```

render_template(*_filename*, ***context*)
Renders a template and returns a response object.

Parameters

- **_filename** – The template filename, related to the templates directory.
- **context** – Keyword arguments used as variables in the rendered template. These will override values set in the request context.

Returns A rendered template.

webapp2_extras.mako.**get_mako**(*factory=<class 'webapp2_extras.mako.Mako'>*, *key='webapp2_extras.mako.Mako'*, *app=None*)
Returns an instance of Mako from the app registry.

It'll try to get it from the current app registry, and if it is not registered it'll be instantiated and registered. A second call to this function will return the same instance.

Parameters

- **factory** – The callable used to build and register the instance if it is not yet registered. The default is the class Mako itself.
- **key** – The key used to store the instance in the registry. A default is used if it is not set.
- **app** – A webapp2.WSGIApplication instance used to store the instance. The active app is used if it is not set.

webapp2_extras.mako.**set_mako**(*mako*, *key='webapp2_extras.mako.Mako'*, *app=None*)
Sets an instance of Mako in the app registry.

Parameters

- **store** – An instance of `Mako`.
- **key** – The key used to retrieve the instance from the registry. A default is used if it is not set.
- **request** – A `webapp2.WSGIApplication` instance used to retrieve the instance. The active app is used if it is not set.

5.7 ProtoRPC

`webapp2_extras.protorpc` makes webapp2 compatible with ProtoRPC services. You can register service mappings in a normal webapp2 WSGI application, and it will be fully compatible with the ProtoRPC library.

Check the ProtoRPC documentation or the ProtoRPC project page for usage details.

Warning: This is an experimental package, as the ProtoRPC API is not stable yet. `webapp2_extras.protorpc` is compatible with the ProtoRPC version shipped with the App Engine SDK (since version 1.5.1).

5.8 Extra routes

This module provides several extra route classes for convenience: domain and subdomain routing, prefixed routes or routes for automatic redirection.

class `webapp2_extras.routes.`**`DomainRoute`**(*template*, *routes*)

A route used to restrict route matches to a given domain or subdomain.

For example, to restrict routes to a subdomain of the appspot domain:

```
app = WSGIApplication([
    DomainRoute('<subdomain>.app-id.appspot.com', [
        Route('/foo', 'FooHandler', 'subdomain-thing'),
    ]),
    Route('/bar', 'BarHandler', 'normal-thing'),
])
```

The template follows the same syntax used by `webapp2.Route` and must define named groups if any value must be added to the match results. In the example above, an extra *subdomain* keyword is passed to the handler, but if the regex didn't define any named groups, nothing would be added.

`__init__`(*template*, *routes*)

Initializes a URL route.

Parameters

- **template** – A route template to match against `environ['SERVER_NAME']`. See a syntax description in `webapp2.Route.__init__()`.
- **routes** – A list of `webapp2.Route` instances.

class `webapp2_extras.routes.`**`RedirectRoute`**(*template*, *handler=None*, *name=None*, *defaults=None*, *build_only=False*, *handler_method=None*, *methods=None*, *schemes=None*, *redirect_to=None*, *redirect_to_name=None*, *strict_slash=False*)

A convenience route class for easy redirects.

It adds redirect_to, redirect_to_name and strict_slash options to webapp2.Route.

__init__(*template*, *handler=None*, *name=None*, *defaults=None*, *build_only=False*, *handler_method=None*, *methods=None*, *schemes=None*, *redirect_to=None*, *redirect_to_name=None*, *strict_slash=False*)

Initializes a URL route. Extra arguments compared to webapp2.Route.__init__():

Parameters

- **redirect_to** – A URL string or a callable that returns a URL. If set, this route is used to redirect to it. The callable is called passing `(handler, *args, **kwargs)` as arguments. This is a convenience to use `RedirectHandler`. These two are equivalent:

```
route = Route('/foo', handler=webapp2.RedirectHandler, defaults={'_uri': '/bar'})
route = Route('/foo', redirect_to='/bar')
```

- **redirect_to_name** – Same as *redirect_to*, but the value is the name of a route to redirect to. In the example below, accessing '/hello-again' will redirect to the route named 'hello':

```
route = Route('/hello', handler=HelloHandler, name='hello')
route = Route('/hello-again', redirect_to_name='hello')
```

- **strict_slash** – If True, redirects access to the same URL with different trailing slash to the strict path defined in the route. For example, take these routes:

```
route = Route('/foo', FooHandler, strict_slash=True)
route = Route('/bar/', BarHandler, strict_slash=True)
```

  Because **strict_slash** is True, this is what will happen:

  - Access to `/foo` will execute `FooHandler` normally.
  - Access to `/bar/` will execute `BarHandler` normally.
  - Access to `/foo/` will redirect to `/foo`.
  - Access to `/bar` will redirect to `/bar/`.

class webapp2_extras.routes.**PathPrefixRoute**(*prefix*, *routes*)

Same as NamePrefixRoute, but prefixes the route path.

For example, imagine we have these routes:

```
app = WSGIApplication([
    Route('/users/<user:\w+>/', UserOverviewHandler, 'user-overview'),
    Route('/users/<user:\w+>/profile', UserProfileHandler, 'user-profile'),
    Route('/users/<user:\w+>/projects', UserProjectsHandler, 'user-projects'),
])
```

We could refactor them to reuse the common path prefix:

```
app = WSGIApplication([
    PathPrefixRoute('/users/<user:\w+>', [
        Route('/', UserOverviewHandler, 'user-overview'),
        Route('/profile', UserProfileHandler, 'user-profile'),
        Route('/projects', UserProjectsHandler, 'user-projects'),
    ]),
])
```

This is not only convenient, but also performs better: the nested routes will only be tested if the path prefix matches.

__init__(*prefix*, *routes*)
Initializes a URL route.

Parameters

- **prefix** – The prefix to be prepended. It must start with a slash but not end with a slash.
- **routes** – A list of `webapp2.Route` instances.

class webapp2_extras.routes.**NamePrefixRoute**(*prefix*, *routes*)
The idea of this route is to set a base name for other routes:

```
app = WSGIApplication([
    NamePrefixRoute('user-', [
        Route('/users/<user:\w+>/', UserOverviewHandler, 'overview'),
        Route('/users/<user:\w+>/profile', UserProfileHandler, 'profile'),
        Route('/users/<user:\w+>/projects', UserProjectsHandler, 'projects'),
    ]),
])
```

The example above is the same as setting the following routes, just more convenient as you can reuse the name prefix:

```
app = WSGIApplication([
    Route('/users/<user:\w+>/', UserOverviewHandler, 'user-overview'),
    Route('/users/<user:\w+>/profile', UserProfileHandler, 'user-profile'),
    Route('/users/<user:\w+>/projects', UserProjectsHandler, 'user-projects'),
])
```

__init__(*prefix*, *routes*)
Initializes a URL route.

Parameters

- **prefix** – The prefix to be prepended.
- **routes** – A list of `webapp2.Route` instances.

class webapp2_extras.routes.**HandlerPrefixRoute**(*prefix*, *routes*)
Same as `NamePrefixRoute`, but prefixes the route handler.

__init__(*prefix*, *routes*)
Initializes a URL route.

Parameters

- **prefix** – The prefix to be prepended.
- **routes** – A list of `webapp2.Route` instances.

5.9 Secure cookies

This module provides a serializer and deserializer for signed cookies.

class webapp2_extras.securecookie.**SecureCookieSerializer**(*secret_key*)
Serializes and deserializes secure cookie values.

Extracted from Tornado and modified.

__init__(*secret_key*)
Initiliazes the serializer/deserializer.

Parameters **secret_key** – A random string to be used as the HMAC secret for the cookie signature.

serialize(*name*, *value*)
Serializes a signed cookie value.

Parameters

- **name** – Cookie name.
- **value** – Cookie value to be serialized.

Returns A serialized value ready to be stored in a cookie.

deserialize(*name*, *value*, *max_age=None*)
Deserializes a signed cookie value.

Parameters

- **name** – Cookie name.
- **value** – A cookie value to be deserialized.
- **max_age** – Maximum age in seconds for a valid cookie. If the cookie is older than this, returns None.

Returns The deserialized secure cookie, or None if it is not valid.

5.10 Security

This module provides security related helpers such as secure password hashing tools and a random string generator.

webapp2_extras.security.**generate_random_string**(*length=None*, *entropy=None*, *pool='ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnop*

Generates a random string using the given sequence pool.

To generate stronger passwords, use ASCII_PRINTABLE as pool.

Entropy is:

H = log2(N**L)

where:

•H is the entropy in bits.

•N is the possible symbol count

•L is length of string of symbols

Entropy chart:

```
-----------------------------------------------------------------
Symbol set               Symbol Count (N)  Entropy per symbol (H)
-----------------------------------------------------------------
HEXADECIMAL_DIGITS       16                4.0000 bits
DIGITS                   10                3.3219 bits
LOWERCASE_ALPHA          26                4.7004 bits
UPPERCASE_ALPHA          26                4.7004 bits
PUNCTUATION              32                5.0000 bits
LOWERCASE_ALPHANUMERIC   36                5.1699 bits
UPPERCASE_ALPHANUMERIC   36                5.1699 bits
ALPHA                    52                5.7004 bits
ALPHANUMERIC             62                5.9542 bits
```

```
ASCII_PRINTABLE         94                6.5546 bits
ALL_PRINTABLE           100               6.6438 bits
```

Parameters

- **length** – The length of the random sequence. Use this or *entropy*, not both.
- **entropy** – Desired entropy in bits. Use this or *length*, not both. Use this to generate passwords based on entropy: http://en.wikipedia.org/wiki/Password_strength
- **pool** – A sequence of characters from which random characters are chosen. Default to case-sensitive alpha-numeric characters.

Returns A string with characters randomly chosen from the pool.

webapp2_extras.security.**generate_password_hash**(*password*, *method='sha1'*, *length=22*, *pepper=None*)

Hashes a password.

The format of the string returned includes the method that was used so that `check_password_hash()` can check the hash.

This method can **not** generate unsalted passwords but it is possible to set the method to plain to enforce plaintext passwords. If a salt is used, hmac is used internally to salt the password.

Parameters

- **password** – The password to hash.
- **method** – The hash method to use (`'md5'` or `'sha1'`).
- **length** – Length of the salt to be created.
- **pepper** – A secret constant stored in the application code.

Returns

A formatted hashed string that looks like this:

```
method$salt$hash
```

This function was ported and adapted from Werkzeug.

webapp2_extras.security.**check_password_hash**(*password*, *pwhash*, *pepper=None*)

Checks a password against a given salted and hashed password value.

In order to support unsalted legacy passwords this method supports plain text passwords, md5 and sha1 hashes (both salted and unsalted).

Parameters

- **password** – The plaintext password to compare against the hash.
- **pwhash** – A hashed string like returned by `generate_password_hash()`.
- **pepper** – A secret constant stored in the application code.

Returns *True* if the password matched, *False* otherwise.

This function was ported and adapted from Werkzeug.

webapp2_extras.security.**hash_password**(*password*, *method*, *salt=None*, *pepper=None*)

Hashes a password.

Supports plaintext without salt, unsalted and salted passwords. In case salted passwords are used hmac is used.

Parameters

- **password** – The password to be hashed.
- **method** – A method from `hashlib`, e.g., *sha1* or *md5*, or *plain*.
- **salt** – A random salt string.
- **pepper** – A secret constant stored in the application code.

Returns A hashed password.

This function was ported and adapted from Werkzeug.

`webapp2_extras.security.`**`compare_hashes`**(*a*, *b*)

Checks if two hash strings are identical.

The intention is to make the running time be less dependant on the size of the string.

Parameters

- **a** – String 1.
- **b** – String 2.

Returns True if both strings are equal, False otherwise.

5.11 Sessions

This module provides a lightweight but flexible session support for webapp2.

It has three built-in backends: secure cookies, memcache and datastore. New backends can be added extending `CustomBackendSessionFactory`.

The session store can provide multiple sessions using different keys, even using different backends in the same request, through the method `SessionStore.get_session()`. By default it returns a session using the default key from configuration.

`webapp2_extras.sessions.`**`default_config`**

Default configuration values for this module. Keys are:

secret_key Secret key to generate session cookies. Set this to something random and unguessable. This is the only required configuration key: an exception is raised if it is not defined.

cookie_name Name of the cookie to save a session or session id. Default is *session*.

session_max_age: Default session expiration time in seconds. Limits the duration of the contents of a cookie, even if a session cookie exists. If None, the contents lasts as long as the cookie is valid. Default is None.

cookie_args Default keyword arguments used to set a cookie. Keys are:

- max_age: Cookie max age in seconds. Limits the duration of a session cookie. If None, the cookie lasts until the client is closed. Default is None.
- domain: Domain of the cookie. To work accross subdomains the domain must be set to the main domain with a preceding dot, e.g., cookies set for *.mydomain.org* will work in *foo.mydomain.org* and *bar.mydomain.org*. Default is None, which means that cookies will only work for the current subdomain.
- path: Path in which the authentication cookie is valid. Default is */*.
- secure: Make the cookie only available via HTTPS.
- httponly: Disallow JavaScript to access the cookie.

backends A dictionary of available session backend classes used by `SessionStore.get_session()`.

class webapp2_extras.sessions.**SessionStore**(*request*, *config=None*)
A session provider for a single request.

The session store can provide multiple sessions using different keys, even using different backends in the same request, through the method `get_session()`. By default it returns a session using the default key.

To use, define a base handler that extends the dispatch() method to start the session store and save all sessions at the end of a request:

```
import webapp2

from webapp2_extras import sessions

class BaseHandler(webapp2.RequestHandler):
    def dispatch(self):
        # Get a session store for this request.
        self.session_store = sessions.get_store(request=self.request)

        try:
            # Dispatch the request.
            webapp2.RequestHandler.dispatch(self)
        finally:
            # Save all sessions.
            self.session_store.save_sessions(self.response)

    @webapp2.cached_property
    def session(self):
        # Returns a session using the default cookie key.
        return self.session_store.get_session()
```

Then just use the session as a dictionary inside a handler:

```
# To set a value:
self.session['foo'] = 'bar'

# To get a value:
foo = self.session.get('foo')
```

A configuration dict can be passed to `__init__()`, or the application must be initialized with the `secret_key` configuration defined. The configuration is a simple dictionary:

```
config = {}
config['webapp2_extras.sessions'] = {
    'secret_key': 'my-super-secret-key',
}

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

Other configuration keys are optional.

__init__(*request*, *config=None*)
Initializes the session store.

Parameters

- **request** – A `webapp2.Request` instance.

- **config** – A dictionary of configuration values to be overridden. See the available keys in `default_config`.

get_backend(*name*)
Returns a configured session backend, importing it if needed.

Parameters **name** – The backend keyword.

Returns A `BaseSessionFactory` subclass.

get_session(*name=None*, *max_age=<object object at 0x2c61eb0>*, *factory=None*, *backend='securecookie'*)
Returns a session for a given name. If the session doesn't exist, a new session is returned.

Parameters

- **name** – Cookie name. If not provided, uses the `cookie_name` value configured for this module.
- **max_age** – A maximum age in seconds for the session to be valid. Sessions store a timestamp to invalidate them if needed. If *max_age* is None, the timestamp won't be checked.
- **factory** – A session factory that creates the session using the preferred backend. For convenience, use the *backend* argument instead, which defines a backend keyword based on the configured ones.
- **backend** – A configured backend keyword. Available ones are:
  - `securecookie`: uses `SecureCookieSessionFactory`. This is the default backend.
  - `datastore`: uses `webapp2_extras.appengine.sessions_ndb.DatastoreSessionFactor`
  - `memcache`: uses `webapp2_extras.appengine.sessions_memcache.MemcacheSessionFac`

Returns A dictionary-like session object.

save_sessions(*response*)
Saves all sessions in a response object.

Parameters **response** – A `webapp.Response` object.

class `webapp2_extras.sessions.`**SessionDict**(*container*, *data=None*, *new=False*)
A dictionary for session data.

get_flashes(*key='_flash'*)
Returns a flash message. Flash messages are deleted when first read.

Parameters **key** – Name of the flash key stored in the session. Default is '_flash'.

Returns The data stored in the flash, or an empty list.

add_flash(*value*, *level=None*, *key='_flash'*)
Adds a flash message. Flash messages are deleted when first read.

Parameters

- **value** – Value to be saved in the flash message.
- **level** – An optional level to set with the message. Default is *None*.
- **key** – Name of the flash key stored in the session. Default is '_flash'.

class `webapp2_extras.sessions.`**SecureCookieSessionFactory**(*name*, *session_store*)
A session factory that stores data serialized in a signed cookie.

Signed cookies can't be forged because the HMAC signature won't match.

This is the default factory passed as the *factory* keyword to `SessionStore.get_session()`.

> **Warning:** The values stored in a signed cookie will be visible in the cookie, so do not use secure cookie sessions if you need to store data that can't be visible to users. For this, use datastore or memcache sessions.

class webapp2_extras.sessions.**CustomBackendSessionFactory**(*name*, *session_store*)
Base class for sessions that use custom backends, e.g., memcache.

webapp2_extras.sessions.**get_store**(*factory=<class 'webapp2_extras.sessions.SessionStore'>*, *key='webapp2_extras.sessions.SessionStore'*, *request=None*)
Returns an instance of `SessionStore` from the request registry.

It'll try to get it from the current request registry, and if it is not registered it'll be instantiated and registered. A second call to this function will return the same instance.

Parameters

- **factory** – The callable used to build and register the instance if it is not yet registered. The default is the class `SessionStore` itself.
- **key** – The key used to store the instance in the registry. A default is used if it is not set.
- **request** – A `webapp2.Request` instance used to store the instance. The active request is used if it is not set.

webapp2_extras.sessions.**set_store**(*store*, *key='webapp2_extras.sessions.SessionStore'*, *request=None*)
Sets an instance of `SessionStore` in the request registry.

Parameters

- **store** – An instance of `SessionStore`.
- **key** – The key used to retrieve the instance from the registry. A default is used if it is not set.
- **request** – A `webapp2.Request` instance used to retrieve the instance. The active request is used if it is not set.

CHAPTER

SIX

API REFERENCE - WEBAPP2_EXTRAS.APPENGINE

App Engine-specific modules.

6.1 Memcache sessions

6.2 Datastore sessions

6.3 Users

CHAPTER

SEVEN

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

CHAPTER

EIGHT

CREDITS

webapp2 is a superset of webapp, created by the App Engine team.

Because webapp2 is intended to be compatible with webapp, the official webapp documentation is valid for webapp2 too. Parts of this documentation were ported from the official documentation for App Engine/Python, written by the App Engine team and licensed under the Creative Commons Attribution 3.0 License.

webapp2 has code ported from Werkzeug and Tipfy.

webapp2_extras has code ported from Werkzeug, Tipfy and Tornado Web Server.

The Sphinx theme mimics the App Engine official documentation.

This library was not created and is not maintained by Google.

CHAPTER

NINE

CONTRIBUTE

webapp2 is considered stable, feature complete and well tested, but if you think something is missing or is not working well, please describe it in our issue tracker:

> http://code.google.com/p/webapp-improved/issues/list

Let us know if you found a bug or if something can be improved. New tutorials and webapp2_extras modules are also welcome, and tests or documentation are never too much.

Thanks!

CHAPTER

TEN

LICENSE

webapp2 is licensed under the Apache License 2.0.

PYTHON MODULE INDEX

W