Chapter 40
Web Services in Python

40.1 Introduction

This chapter looks at RESTful web services as implemented using the Flask framework.

40.2 RESTful Services

REST stands for Representational State Transfer and was a termed coined by Roy Fielding in his Ph.D. to describe the lightweight, resource-oriented architectural style that underpins the web. Fielding, one of the principle authors of HTTP, was looking for a way of generalising the operation of HTTP and the web. The generalised the supply of web pages as a form of data supplied on demand to a client where the client holds the current state of an exchange. Based on this state information the client requests the next item of relevant data sending all information necessary to identify the information to be supplied with the request. Thus the requests are independent and not part of an on-going stateful conversation (hence state transfer).

It should be noted that although Fielding was aiming to create a way of describing the pattern of behaviour within the web, he also had an eye on producing lighter weight web based services (than those using either proprietary Enterprise Integration frameworks or SOAP based services). These lighter weight HTTP based web services have become very popular and are now widely used in many areas. Systems which follow these principles are termed RESTful services.

A key aspect of a RESTful service is that all interactions between a client (whether some JavaScript running in a browser or a standalone application) are done using simple HTTP based operations. HTTP supports four operations these are HTTP Get, HTTP Post, HTTP Put and HTTP Delete. These can be used as

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-25943-3_40

verbs to indicate the type of action being requested. Typically these are used as follows:

- retrieve information (HTTP Get),
- create information (HTTP Post),
- update information (HTTP Put),
- delete information (HTTP Delete).

It should be noted that REST is not a standard in the way that HTML is a standard. Rather it is a design pattern that can be used to create web applications that can be invoked over HTTP and that give meaning to the use of Get, Post, Put and Delete HTTP operations with respect to a specific resource (or type of data).

The advantage of using RESTful services as a technology, compared to some other approaches (such as SOAP based services which can also be invoked over HTTP) is that

- the implementations tend to be simpler,
- the maintenance easier,
- they run over standard HTTP and HTTPS protocols and
- do not require expensive infrastructures and licenses to use.

This means that there is lower server and server side costs. There is little vendor or technology dependency and clients do not need to know anything about the implementation details or technologies being used to create the services.

40.3 A RESTful API

1. A RESTful API is one in which you must first determine the key concepts or *resources* being represented or managed.
2. These might be books, products in a shop, room bookings in hotels etc. For example a bookstore related service might provide information on resources such as books, CDs, DVDs, etc. Within this service books are just one type of resource. We will ignore the other resources such as DVDs and CDs etc.
3. Based on the idea of a book as a resource we will identify suitable URLs for these RESTful services. Note that although URLs are frequently used to describe a web page—that is just one type of resource. For example, we might develop a resource such as

   ```
   /bookservice/book
   ```

   from this we could develop a URL based API, such as

   ```
   /bookservice/book/<isbn>
   ```

Where ISBN (the International Standard Book Number) indicates a unique number to be used to identify a specific book whose details will be returned using this URL.

We also need to design the representation or formats that the service can supply. These could include plain text, JSON, XML etc. JSON standards for the JavaScript Object Notation and is a concise way to describe data that is to be transferred from a service running on a server to a client running in a browser. This is the format we will use in the next section. As part of this we might identify a series of operations to be provided by our services based on the type of HTTP Method used to invoke our service and the contents of the URL provided. For example, for a simple BookService this might be:

- GET `/book/<isbn>`—used to retrieve a book for a given ISBN.
- GET `/book/list`—used to retrieve all current books in JSON format.
- POST `/book` (JSON in body of the message)—which supports creating a new book.
- PUT `/book` (JSON in body of message)—used to update the data held on an existing Book.
- DELETE `/book/<isbn>`—used to indicate that we would like a specific book deleted from the list of books held.

Note that the *parameter* `isbn` in the above URLs actually forms part of the URL path.

40.4 Python Web Frameworks

There are very many frameworks and libraries available in Python that will allow you to create JSON based web services; and the shear number of options available to you can be overwhelming. For example, you might consider

- Flask,
- Django,
- Web2py and
- CherryPy to name just a few.

These frameworks and libraries offer different sets of facilities and levels of sophistication. For example Django is a full-stack web framework; that is it is aimed at developing not just web services but full blown web sites.

However, for our purposes this is probably overkill and the Django Rest interface is only part of a much larger infrastructure. That does not mean of course that we could not use Django to create our bookshop services; however there are simpler options available. The web2py is another full stack web framework which we will also discount for the same reason.

In contrast Flask and CherryPy are considered non full-stack frameworks (although you can create a full stack web application using them). This means that they are lighter weight and quicker to get started with. CherryPy was original rather more focussed on providing a remote function call facility that allowed functions to be invoked over HTTP; however this has been extended to provide more REST like facilities.

In this chapter we will focus on Flask as it is one of the most widely used frameworks for light weight RESTful services in Python.

40.5 Flask

Flask is a web development framework for Python. It describes itself as a *micro framework* for Python which is somewhat confusing; to the point where there is a page dedicated to this on their web site that explains what it means and what the implications are of this for Flask. According to Flask, the *micro* in its description relates to its primary aim of keeping the core of Flask simple but extensible. Unlike Django it doesn't include facilities aimed at helping you integrate your application with a database for example. Instead Flask focuses on the core functionality required of a web service framework and allows extension to be used, as and when required, for additional functionality.

Flask is also a convention over configuration framework; that is if you follow the standard conventions then you will not need to deal with much additional configuration information (although if you wish to follow a different set of conventions then you can provide configuration information to change the defaults). As most people will (at least initially) follow these conventions it makes it very easy to get something up and running very quickly.

40.6 Hello World in Flask

As is traditional in all programming languages we will start of with a simple 'Hello World' style application. This application will allow us to create a very simple web service that maps a particular URL to a function that will return JSON format data. We will use the JSON data format as it is very widely used within web-based services.

40.6.1 Using JSON

JSON standards for JavaScript Object Notation; it is a light weight data-interchange format that is also easy for humans to read and write. Although it is derived from a subset of the JavaScript programming language; it is in fact completely language independent and many languages and frameworks now support automatically processing of their own formats into and from JSON. This makes it ideal for RESTful web services.

JSON is actually built on some basic structures:

- A collection of name/value pairs in which the name and value are separated buy a colon ':' and each pair can be separated by a comma ','.
- An ordered list of values that are encompassed in square brackets ('[]').

This makes it very easy to build up structures that represent any set of data, for example a book with an ISBN, a title, author and price could be represented by:

```
{
      "author": "Phoebe Cooke",
      "isbn": 2,
      "price": 12.99,
      "title": "Java"
}
```

In turn a list of books can be represented by a comma separated set of books within square brackets. For example:

```
[ {"author": "Gryff Smith","isbn": 1, "price": 10.99, "title":
       "XML"},
 {"author": "Phoebe Cooke", "isbn": 2, "price": 12.99, "title":
       "Java"}
{"author": "Jason Procter", "isbn": 3, "price": 11.55, "title":
       "C#"}]
```

40.6.2 Implementing a Flask Web Service

There are several steps involved in creating a Flask web service, these are:

1. Import flask.
2. Initialise the Flask application.
3. Implement one or more functions (or methods) to support the services you wish to publish.
4. Providing routing information to route from the URL to a function (or method).
5. Start the web service running.

We will look at these steps in the rest of this chapter.

40.6.3 A Simple Service

We will now create our *hello world* web service. To do this we must first import the flask module. In this example we will use the `Flask` class and `jsonify()` function elements of the module.

We then need to create the main application object which is an instance of the `Flask` class:

```
from flask import Flask, jsonify

app = Flask(__name__)
```

The argument passed into the `Flask()` constructor is the name of the application's module or package. As this is a simple example we will use the `__name__` attribute of the module which in this case will be '`__main__`'. In larger more complex applications, with multiple packages and modules, then you may need to choose an appropriate package name.

The Flask application object implements the WSGI (Web Server Gateway Interface) standard for Python. This was originally specified in PEP-333 in 2003 and was updated for Python 3 in PEP-3333 published in 2010. It provides a simple convention for how web servers should handle requests to applications. The Flask application object is the element that can route a request for a URL to a Python function.

40.6.4 Providing Routing Information

We can now define routing information for the Flask application object. This information will map a URL to a function. When that URL is, for example, entered into a web browsers URL field, then the Flask application object will receive that request and invoke the appropriate function.

To provide route mapping information we use the `@app.route` decorator on a function or method.

For example, in the following code the `@app.route` decorator maps the URL `/hello` to the function `welcome()` for HTTP Get requests:

```
@app.route('/hello', methods=['GET'])
def welcome():
    return jsonify({'msg': 'Hello Flask World'})
```

There are two things to note about this function definition:

- The `@app.route` decorator is used to declaratively specify the routing information for the function. This means that the URL '`/hello`' will be mapped to the function `welcome()`. The decorator also specifies the HTTP method that is supported; in this case `GET` requests are supported (which is actually the default so it does not need to be included here but is useful from a documentation point of view).

- The second thing is that we are going to return our data using the JSON format; we therefore use the `jsonify()` function and pass it a Python Dictionary structure with a single key/value pair. In this case the key is ‘`msg`’ and the data associated with that key is ‘`Hello Flask World`’. The `jsonify()` function will convert this Python data structure into an equivalent JSON structure.

40.6.5 Running the Service

We are now ready to run our application. To do this we invoke the `run()` method of the `Flask` application object:

```
app.run(debug=True)
```

Optionally this method has a keyword parameter `debug` that can be set to `True`; if this is done then when the application is run some debugging information is generated that allows you to see what is happening. This can be useful in development but would not typically be used in production.

The whole program is presented below:

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/hello', methods=['GET'])
def welcome():
    return jsonify({'msg': 'Hello Flask World'})

app.run(debug=True)
```

When this program is run the initial output generated is as shown below:

```
 * Serving Flask app "hello_flask_world" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a
production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 274-630-732
```

Of course we don’t see any output from our own program yet. This is because we have not invoked the `welcome()` function via the `/hello` URL.

40.6.6 *Invoking the Service*

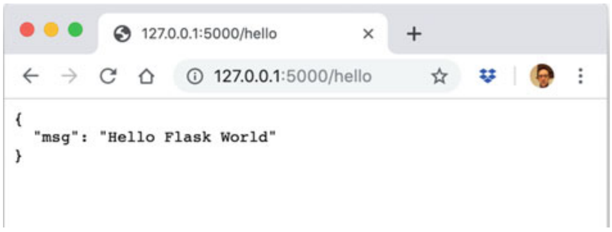
We will use a web browser to access the web service. To do this we must enter the full URL that will route the request to our running application and to the `welcome()` function.

The URL is actually comprised of two elements, the first part is the machine on which the application is running and the port that it is using to listen for requests. This is actually listed in the above output—look at the line starting ‘Running on’. This means that the URL must start with `http://127.0.0.1:5000`. This indicates that the application is running on the computer with the IP address `127.0.0.1` and listening on port `5000`.

We could of course also use `localhost` instead of `127.0.0.1`.

The remainder of the URL must then provide the information that will allow Flask to route from the computer and port to the functions we want to run.

Thus the full URL is `http://127.0.0.1:5000/hello` and thus is used in the web browser shown below:



```
{
  "msg": "Hello Flask World"
}
```

As you can see the result returned is the text we supplied to the `jsonify()` function but now in plain JSON format and displayed within the Web Browser.

You should also be able to see in the console output that a request was received by the Flask framework for the GET request mapped to the `/hello` URL:

```
127.0.0.1 - - [23/May/2019 11:09:40] "GET /hello HTTP/1.1" 200
-
```

One useful feature of this approach is that if you make a change to your program then the Flask framework will notice this change when running in development mode and can restart the web service with the code changes deployed. If you do this you will see that the output notifies you of the change:

```
* Detected change in 'hello_flask_world.py', reloading
* Restarting with stat
```

This allows changes to be made on the fly and their effect can be immediately seen.

40.6.7 The Final Solution

We can tidy this example up a little by defining a function hat can be used to create the `Flask` application object and by ensuring that we only run the application if the code is being run as the `main` module:

```
from flask import Flask, jsonify, url_for

def create_service():
    app = Flask(__name__)

    @app.route('/hello', methods=['GET'])
    def welcome():
        return jsonify({'msg': 'Hello Flask World'})

    with app.test_request_context():
        print(url_for('welcome'))

    return app

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

One feature we have added to this program is the use of the `test_request_context()`. The test request context object returned implements the context manager protocol and thus can be used via a `with` statement; this is useful for debugging purposes. It can be used to verify the URL used for any functions with routing information specified. In this case the output from the print statement is '/hello' as this is the URL defined by the `@app.route` decorator.

40.7 Online Resources

See the following online resources for information on the topics in this chapter:

- http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm Roy Fieldings' Ph.D. Thesis; if you are interesting in the background to REST read this.
- https://wiki.python.org/moin/WebFrameworks for a very extensive list of web frameworks for Python.
- https://www.djangoproject.com/ for information on Django.
- http://www.web2py.com/ Web2py web framework documentation.
- https://cherrypy.org/ For documentation on the CherryPy web framework.
- http://flask.pocoo.org/ For information and examples on the Flask web development framework.

- http://flask.pocoo.org/docs/1.0/foreword/#what-does-micro-mean Flasks explanation of what micro means.
- https://www.json.org/ Information on JSON.
- https://en.wikipedia.org/wiki/Web_Server_Gateway_Interface WSGI Web Server Gateway Interface standard.
- https://curl.haxx.se/ Information on the curl command line tool.
- https://developer.mozilla.org/en-US/docs/Web/HTTP/Status HTTP Response Status Codes.