Edinburgh Napier University

SET09103 - Advanced Web Technologies

Topic 07 - APIs & Web Services

Dr Simon Wells

1 Overview

GOAL: Our aim in this practical is to elaborate on our simple Flask web-app up from before to increase our ability to use different features from HTTP and to take advantage of features that Flask might have. In this topic, that means specifically the following:

1. Creating a data API.
2. Returing JSON instead of HTML.

Again, the 'Hello World' web-app from the first topic will be the foundation for the majority of the work that we do in the rest of the trimester so it is important to be comfortable with it. If you're still unsure, try it again from the beginning and consider asking a demonstrator to explain things a little.

Don't just go through the practical once and then move on. Instead, try the following approach:

1. Go through the practical once exactly as presented.
2. Go through the lab at least once more so that you are used to everything involved. We will build on the knowledge from this lab and assume it in subsequent topics so get comfortable with it before moving on.

The practicals are built around the following basic structure:

1. An introduction to give you a high-level overview.
2. Some exercise activities that you should iterate through until **you** feel confident with the material covered.
3. Some challenges to give you a starting place for play and exploration.
4. Some additional resources & documentation to help you find more information and start filling in any gaps.

IMPORTANT: If something doesn't work, or you're not sure whether something has worked, then stop for a moment, think and consider what output you are aiming for rather than just plowing on. If you move ahead before understanding something then, if you ahve made a mis-step, you will be compounding error upon error, making it more difficult to extricate yourself. A good rule of thumb when coding is to *aim to always keep things working*. So if something breaks, or doesn't give you the result you want, you should unwind things back to a *known good state*.

2 Activities: Exercises

Previous topics have enabled us to create hierarchies of addresses that relate to web pages in our site. In this topic we'll consider how to create resources that will return data as JSON documents rather than HTML. This will enable us to construct data APIs.

2.1 Getting Started

Log into your lab machine. For this practical you will need a web browser, a text editor, the windows terminal (cmd.exe), and the file manager open as you will probably use a mix of all of them at various points. You will also need uv installed as we covered in the previous lab as well as your working hello world Flask app as a starting point. Check that uv[1] works before proceeding.

2.2 A simple Data API

In this topic we're going to build a simpleSerendipity is the occurence and development of events by chance in a happy or beneficial way. Sometimes, when creating things, or uncertain, we can attempt to harness serendipity to help us to side-step indecision. This inspires the example project that we'll build in this chapter, a simple site to provide us with cues to help us to make decisions when we're unsure. We'll use this as a scenario for developing an API including a small example user interface and an HTTP/JSON interface.

[1] You might need to add it to your path again.

Our two main sources of serendipity in this project will be the 'Magic 8 Ball' and Brian Eno's 'Oblique Strategies'. The magic 8 ball is a toy to which you direct a question, and the result is some random response from a list of supportive, neutral, or dismissive responses. The oblique strategies are fairly similar, but are intended to help you to find creative ways around an impasse, perhaps in a design or other creative endeavour. Whilst the magic 8 ball started out as a toy, the oblique strategies came from musicians who were looking for interesting and creative ways to solve problems and make interesting sounds.

From the Flask perspective, the aim of this chapter is to explore some elements involved in building a simple HTTP based API. We'll create a few routes and associated functions, that will demonstrate:

1. How to build a simple JSON API
2. How to build a simple form-based UI
3. How to integrate flask routes and non-flask Python functions

2.3 The Magic Eight Ball Web-App

This is meant to illustrate how we can use a route, the query route, to respond to multiple HTTP methods and to return JSON to the client. This uses the Flask jsonify method to build a JSON response instead of an HTML response. Let's see it in action:

```
from flask import Flask, request, jsonify
import random, json

app = Flask(__name__)

@app.route("/query/", methods=['GET','POST'])
def query():

    if request.method == 'POST':

        question = request.get_json().get("question")

        data = {
            "question": question,
            "answer": magic8ball()
        }
        return jsonify(data), 200

    else:
        data = {
            "answer": magic8ball()
        }

        return jsonify(data), 200

def magic8ball():
    responses = [
        "It is certain.",
        "It is decidedly so.",
        "Without a doubt.",
        "Yes definitely.",
        "You may rely on it.",
        "As I see it, yes.",
        "Most likely.",
        "Outlook good.",
        "Yes.",
        "Signs point to yes.",
      "Reply hazy, try again.",
        "Ask again later.",
        "Better not tell you now.",
        "Cannot predict now.",
        "Concentrate and ask again.",
        "Don't count on it.",
        "My reply is no.",
        "My sources say no.",
        "Outlook not so good.",
        "Very doubtful."
    ]
    return random.choice(responses)
```

The magic8ball Flask webapp defines two functions. Of these, one of the functions is also a Flask route and the other function is a pure Python function that is called and utilised from the Flask route For example, the magic8ball() function is called by the query() function, which in turn is invoked when the */queryl/* route is called by a client, such as a browser, making a request to that specific route. Take a moment to familiarise yourself with the overall structure of the code, the different functions and how the functions that are also web routes are delineated from those that are not.

Let's start with the *magic8ball* function. This holds a list called *responses* containing the classic responses from a magic eight ball toy stored as string. When the function is called, the random.choice method from the Python random library is applied to the responses list to randomly select and return one of the list items. This random item is returned to the caller.

The */query/* route is our caller of the *magic8ball* function and responds to both HTTP GET and HTTP POST requests. So it requires an if clause to immediately determine whether the incoming request is a GET or a POST so that it can treat them differently. If the request is a GET then it creates a dictionary containing a single key:value pair. The key is "answer" and the value is one randomly selected magic 8 ball response returned by a call to the magic8ball() function. This dictionary is then turned into a HTTP response containing a JSON string as it's payload using the Flask jsonify function. POST is similar but has some additional functionaliy. This POST request includes a POST'ed JSON request body that contains a single key:value pair { "question":"*question content*" }. The value is extracted from this pair, using the request.get_json() function, and is used to add an extra key"value pair to the data dictionary alongside the magic8ball response. To summarise, we're extracting the question from the incoming POST request, adding the question to our data dictionary, getting a random magic 8 ball answer and adding that to our data dictionary, then turning the entire dictionary into JSON and returning it to the client in our response.

Note that there is a second way of organising your API code that gives the same result but can be easier to maintain once your the functions assocaited with your individual routes grows:

```
from flask import Flask, request, jsonify
import random, json

app = Flask(__name__)

@app.route("/query/", methods=['GET'])
def answer():
    data = {
        "answer": magic8ball()
    }
    return jsonify(data), 200

@app.route("/query/", methods=['POST'])
def question():
    question = request.get_json().get("question")

    data = {
        "question": question,
        "answer": magic8ball()
    }
    return jsonify(data), 200

def magic8ball():
    responses = [
        "It is certain.",
        "It is decidedly so.",
        "Without a doubt.",
        "Yes definitely.",
        "You may rely on it.",
        "As I see it, yes.",
        "Most likely.",
        "Outlook good.",
        "Yes.",
        "Signs point to yes.",
      "Reply hazy, try again.",
        "Ask again later.",
        "Better not tell you now.",
        "Cannot predict now.",
```

```
        "Concentrate and ask again.",
        "Don't count on it.",
        "My reply is no.",
        "My sources say no.",
        "Outlook not so good.",
        "Very doubtful."
    ]
    return random.choice(responses)
```

This second layout is functionally the same as the previous magic8ball example, but instead of using an if-else clause to process the different HTTP methods, we this time use a differnt function for each. So we now have the answer and query functions which both respond to the same route, but one is matched to a GET request and the other is matched to a POST request. This removes a bit of code from our implementation but now has an increased number of functions (which will grow at a rate proportional to the number of HTTP methods that each of your routes recognise.

You can interact with your API through your browser in most modern browsers, at least for GET requests. But to fully explore your API in terms of other HTTP methods you'll need to resort to a tool like cURL to help you craft your calls to your API's routes.

Either approach is entirely valid and which you choose should depend upon the size and complexity of your code and overall API.

2.4 A CRUD API

In this example we'll develop a simple Create Read Update Delete (CRUD) style API for various types of information associated with a collection of superheroes. We'll start with an API for retrieving information:

```
from flask import Flask, request, jsonify
import random, json

app = Flask(__name__)

data = {
        "Captain Marvel":{
            "hero name":"Captain Marvel",
            "full name":"Carol Danvers",
            "teams":["Avengers", "Anachronauts"]
            },
        "Hulk":{
            "hero name":"Hulk",
            "full name":"Robert Bruce Banner",
            "teams":["Avengers", "Defenders", "Marvel Knights"]
            },
        "Beast":{
            "hero name":"Beast",
            "full name": "Henry Philip McCoy",
            "teams":["Avengers", "X-Men"]
            }
    }


@app.route("/heroes/", methods=['GET'])
def heroes():
    heroes = list(data.keys())
    return jsonify(heroes), 200

@app.route("/heroes/<name>/", methods=['GET'])
def name(name):
    hero = data[name]
    return jsonify(hero), 200

@app.route("/heroes/<name>/hero-name/", methods=['GET'])
def hero_name(name):
    heroname = data[name]['hero name']
    return jsonify(heroname), 200

@app.route("/heroes/<name>/full-name/", methods=['GET'])
def full_name(name):
```

```
    fullname = data[name]['full name']
    return jsonify(fullname), 200

@app.route("/heroes/<name>/teams/", methods=['GET'])
def teams(name):
    teams = list(data[name]['teams'])
    return jsonify(teams), 200
```

This is just a simple dataset, stored using a Python dictionary called 'data', which could be easily replaced by a call to a database or serialised to an external file. However, for clarity and simplicity in this example, a dictionary suffices.

A series of routes is implemented that iteratively explore the data's hierarchy, enabling either collections of information to be returned, or individual data items. The *heroes* and *teams* routes return collections of information, in the form of lists, whilst the other routes return individual datapoint or structured groups of data, e.g. */hero/name* returns all the information about that hero whereas */hero/name/full-name* returns the hero's full non hero name.

This next version demonstrates how a POST route can be used to add a new entry to the data object. In this first version we'll use an HTML form as the way to supply the new data, then later we'll show how to use an HTTP POST request where a JSON document is passed in containing the new data to add.

```
from flask import Flask, request, jsonify
import random, json

app = Flask(__name__)

data = {
        "Captain Marvel":{
            "hero name":"Captain Marvel",
            "full name":"Carol Danvers",
            "teams":["Avengers", "Anachronauts"]
            },
        "Hulk":{
            "hero name":"Hulk",
            "full name":"Robert Bruce Banner",
            "teams":["Avengers", "Defenders", "Marvel Knights"]
            },
        "Beast":{
            "hero name":"Beast",
            "full name": "Henry Philip McCoy",
            "teams":["Avengers", "X-Men"]
            }
    }

@app.route("/heroes/", methods=['POST'])
def query():
    newhero = {}
    heroname = request.form['hero-name']
    newhero['hero name'] = heroname
    newhero['full name'] = request.form['full-name']
    teams = request.form['teams']
    newhero['teams'] = [x.strip() for x in teams.split(',')]
    data[heroname] = newhero
    return (jsonify(data))

@app.route("/heroes/", methods=['GET'])
def heroes():
    heroes = list(data.keys())
    return jsonify(heroes), 200

@app.route("/heroes/<name>/", methods=['GET'])
def name(name):
    hero = data[name]
    return jsonify(hero), 200

@app.route("/heroes/<name>/hero-name/", methods=['GET'])
def hero_name(name):
    heroname = data[name]['hero name']
    return jsonify(heroname), 200
```

```
49
50 @app.route("/heroes/<name>/full-name/", methods=['GET'])
51 def full_name(name):
52     fullname = data[name]['full name']
53     return jsonify(fullname), 200
54
55 @app.route("/heroes/<name>/teams/", methods=['GET'])
56 def teams(name):
57     teams = list(data[name]['teams'])
58     return jsonify(teams), 200
59
60 @app.route("/helper/", methods=['GET'])
61 def helper():
62     page='''
63         <!DOCTYPE html>
64         <html><body>
65             <form action="/heroes/" method="post" name="form">
66
67                 <label for="hero-name">Hero Name:</label>
68                 <input type="text" name="hero-name" id="hero-name"/>
69
70                 <label for="full-name">Full Name:</label>
71                 <input type="text" name="full-name" id="full-name"/>
72
73                 <label for="teams">Teams (comma separated list):</label>
74                 <input type="text" name="teams" id="teams"/>
75
76                 <input type="submit" name="submit" id="submit"/>
77             </form>
78         <body></html>
79     '''
80     return page
```

Sometimes it can be useful to be able to interact with an API from a web page. So the last example demonstrated how a helper route and function was used to create an HTML form that then POST'ed to our site. A new hero dictionary was added to our data dictionary which was then available through the rest of the API. Note that this doesn't persist any POST'ed changes, so if you restart your flask app then you'll lose any updates. For persistence you'll need to either save your data dictionary to a file or else replace the data dictionary with a database.

```
1  from flask import Flask, request, jsonify
2  import random, json
3
4  app = Flask(__name__)
5
6  data = {
7          "Captain Marvel":{
8              "hero name":"Captain Marvel",
9              "full name":"Carol Danvers",
10             "teams":["Avengers", "Anachronauts"]
11             },
12         "Hulk":{
13             "hero name":"Hulk",
14             "full name":"Robert Bruce Banner",
15             "teams":["Avengers", "Defenders", "Marvel Knights"]
16             },
17         "Beast":{
18             "hero name":"Beast",
19             "full name": "Henry Philip McCoy",
20             "teams":["Avengers", "X-Men"]
21             }
22     }
23
24 @app.route("/heroes/", methods=['POST'])
25 def query():
26     content = request.get_json()
27     if(content):
28         print(content)
29         newhero = {}
30         heroname = content['hero-name']
31         newhero['hero name'] = heroname
32         newhero['full name'] = content['full-name']
```

```
        teams = content['teams']
        newhero['teams'] = [x.strip() for x in teams.split(',')]
        data[heroname] = newhero
        print(newhero)
        return(jsonify(data))
    else:
        return (jsonify(data))

@app.route("/heroes/", methods=['GET'])
def heroes():
    heroes = list(data.keys())
    return jsonify(heroes), 200

@app.route("/heroes/<name>/", methods=['GET'])
def name(name):
    hero = data[name]
    return jsonify(hero), 200

@app.route("/heroes/<name>/hero-name/", methods=['GET'])
def hero_name(name):
    heroname = data[name]['hero name']
    return jsonify(heroname), 200

@app.route("/heroes/<name>/full-name/", methods=['GET'])
def full_name(name):
    fullname = data[name]['full name']
    return jsonify(fullname), 200

@app.route("/heroes/<name>/teams/", methods=['GET'])
def teams(name):
    teams = list(data[name]['teams'])
    return jsonify(teams), 200
```

Now that we no longer have the helper web page to provide JSON from our form, we must call our Flask app using cURL, e.g.

```
$ curl -X POST -H "Content-Type: application/json" -d '{"hero-name": "lkdsjf", "full-name":
    "lkdsajf", "teams": "lkjsdf,kldsjf", "submit": "Submit Query"}' http://localhost:5000/
    heroes/
```

Our final example is a different web-app. This time we are creating a simple status API. This allows us to track whether something is good or bad. We're using a Python list to store the value "good" or "bad" sequentially in the list. We've then implemented one route that responds to GET, PUT, POST, and DELETE methods to, variously, GET the entire list, PUT a value in a specific location, POST a new value to the end of the list, and to DELETE all entries in the list. Let's see it in action:

```
from flask import Flask, request, jsonify
import json

app = Flask(__name__)

status_data = []

@app.route("/", methods=['GET'])
def get_status():
    print(status_data)
    return (jsonify(status_data))

@app.route("/", methods=['POST'])
def post_status():
    content = request.get_json()
    if(content):
        status_data.append(content)
    return (jsonify(status_data))


@app.route("/", methods=['PUT'])
def put_status():
    content = request.get_json()
    if(content):
        index = content['index']
        status = content['status']
```

```
27          status_data[index] = status
28      return(jsonify(status_data))
29
30  @app.route("/", methods=['DELETE'])
31  def delete_status():
32      content = request.get_json(silent=True)
33      status_data.clear()
34      return (jsonify(status_data))
```

I expect that you can already see some places where this could be improved. For example, a simple good or bad doesn't tell us too much, but as soon as you add timestamps you have a simple API for collecting time series data which is core to a lot of scientific and monitoring processes. In reality we would also do a lot of checking over the incoming requests, ensuring that any incoming JSON contained only what was needed and was correctly formed[2] otherwise it will most likely break as soon as some JSON is passed to it that doesn't contain the data it expects formatted the way it wants it.

There is a lot more to developing good APIs, and you'll develop skills for doing so as you build more APIs for yourself. But this should have given you the basics for actually implementing routes that responsd in different ways to incoming HTTP requests.

3 Activities: Challenges

Taking any of the Flask apps that you've created so far, including those you made during earlier labs, as a starting point, or else creating something from scratch:

1. Design and implement a simple API to organise and provide access to information sourced from the CIA World Fact Book[3]. Your API should return information about the nominated country. You should design your resource hierarchy so that the caller can retrieve different kinds of information directly by accessing different resources.

2. Extend your world factbook API from the last challenge to use a database to hold the data that you return as JSON.

3. Build a recipe collection API. Consider how the you might use the resources and the associated address hierachy to organise the recipes and enable them to be found in different ways. For example, a recipe resource might be navigated to as a result of exploring a list of ingredients, or a list of dishes for different parts of a meal (starter, main, dessert, etc.) or by nation. You might want to use a database for this once more (which gives you additional experience of both developing an API as well as developing a database.

4. Take some topic, pastime, interest, or hobby of yours and design and implement an API to hold information about your choice.

If you're still not confident with creating a new Flask web-app from scratch[4] don't just edit and re-use your existing ones. The setting up aspect is still part of the practise. For each of the exercises, create a new project folder in your set09103 folder. You will need to create a new virtual env each time and install pip. Each project folder should have a unique name but don't call it app or flask as this will cause errors. Doing this will give you practise in setting up new Flask projects so that it becomes straightforward. We'll be doing that a lot over the coming weeks so making it less of a chore is a good idea. One tactic that can help is to create yourself a crib sheet containing the *invariant* commands that you use each time you set up a new Flask app for development. Over a short period of time, and a few repetitions of practise, you **will** get past this.

If you are confident in creating a new Flask app as needed then it is fine to reuse your existing web-apps as a skeleton for new activities and challenges. Note also that once your have set up a virtualenv it can be reused for different Flask apps, you just need to pass the name of the flask app python file to the "*uv run ...*" invocation to tell uv that you want to run a different flask app.

[2] At which point we probably start to think about documenting our API and the data it consumes as well as using a schema to describe it exactly.

[3] `https://www.cia.gov/the-world-factbook/`

[4] you should be able to do the entire pprocess from the first lab topic in less than five minutes by now.

4 Finally...

If you've forgotten what you can do with HTML and want a little more practise or to look up the details of some tags then see the following:

- HTML Tutorial: `https://www.w3schools.com/html/`
- HTML Exercises: `https://www.w3schools.com/html/html_exercises.asp`
- HTML Element Reference: `https://www.w3schools.com/tags/`

Similarly, if you aren't yet entirely comfortable with Python then see the following:

- Python Tutorial: `https://www.w3schools.com/python/default.asp`
- Python Language Reference: `https://www.w3schools.com/python/python_reference.asp`

Use the following to find out more about HTTP verbs and status codes:

- HTTP Verbs: `https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Methods`
- HTTP Status Codes: `https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Status`

Remember that the best way to become really familiar with a technology is to just use it. The more you use it the easier it gets. Building lots of "throw away" experimental apps, scripts, and sites is a good way to get to achieve this so let your imagination have free reign.