Edinburgh Napier University

SET09103 - Advanced Web Technologies

Topic 05 - Data

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. Sessions
2. Configuration
3. Message Flashing
4. Logging
5. Testing
6. HTML Headers

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

The last topic added both HTTP methods and HTTP status codes to our collection of tools for building our own sites. So we should now be able to create a simple working web app that incorporates multiple routes to our resources, and enables us to interact with them using different methods and to return results with appropraite codes. However, we are still using quoted strings to hold the content of our Web-pages. What we need is a better mechanism to store and return Web pages as regular HTML files. Once we can do that, we'll add other kinds of static file based resources, before adding Jinja2 code to our HTML files to enable them to be dynamically generated with different content at run-time.

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 Configuration & Config Files

Quite often we need to do external configuration of our Flask app, for example, if we are using external tools, like Babel, to control translations into different languages, if we need to specify an encryption key for setting up secure sessions, or a password for accessing an external mail server. We can achieve this by using configuration files, external text files that Flask can read at start up and which provide data to Flask about the environment it is running in.

Create a sub-directory called 'etc' just like you did for the static and templates directories. Your config files will live here. Now create a new text file in the etc directory called 'defaults.cfg', this will be the default file that your flask app reads in at start up. We can now use that file to store some configuration data. Let's start by moving our debug and host settings into the config file instead of setting them in code, e.g. Add the following to your defaults.cfg:

```
[config]
debug =  True
ip_address = 0.0.0.0
port = 5000
url = http://127.0.0.1:5000
```

Our config file can be used to store whichever configuration values we decide we want to support in a flask app. For illustration purposes I have used the debug flag and IP, URL, and port numbers settings. We can now create a Flask app that uses those values, for example,

```
import configparser

from flask import Flask

app = Flask(__name__)

def init(app):
    config = configparser.ConfigParser()
    try:
        print("INIT FUNCTION")
        config_location = "etc/defaults.cfg"
        config.read(config_location)

        app.config['DEBUG'] = config.get("config", "debug")
        app.config['ip_address'] = config.get("config", "ip_address")
        app.config['port'] = config.get("config", "port")
        app.config['url'] = config.get("config", "url")
    except:
        print ("Could not read configs from: ", config_location)

init(app)

@app.route('/')
def root():
  return "Hello Napier from the configuration testing app"

@app.route('/config/')
def config():
  s = []
  s.append('debug:'+str(app.config['DEBUG']))
  s.append('port:'+str(app.config['port']))
  s.append('url:'+str(app.config['url']))
  s.append('ip_address:'+str(app.config['ip_address']))
  return ', '.join(s)
```

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

Notice how we are 'getting' the value for each key from the config file then storing those values in the app.config object.

2.3 Sessions

Sessions are a way to manage user data between requests by storing small amounts of data within a cookie. Cookies are small data files that are stored in the users browser and are suitable for small amounts of, ideally non-private, data. Sessions rely on cookies that are cryptographically secured by Flask using a secret key to ensure that the content of the cookies hasn't been altered by any process other than the Flask app that created it. However, the content of a cookie can easily be read by the client or whilst it is transmitted between the client and server during a request[2].

Flask provides an interface for using 'secured cookies' or *sessions* rom our Python code and we can consider a session to be a small data store for keys and value, a form of dictionay. As a result we can set data into a session, query that data, and remove that data,

To add a key value pair we merely treat the session object as a Python dictionary[3], e.g.

```
# Set key=value, name=simon into a session
session['name'] = simon
```

You can then retrieve the value set for name in the session but because this key might not exist in the session we need to wrap everything in a try-except structure to catch the KeyError that might be raised[4].

```
try:
    if(session['name']):
        return str(session['name'])
except KeyError:
    pass
```

The only other thing that we might need to do is to remove a specified key and it's associated value from the session. We do this using the pop function, e.g.

```
session.pop('name', None)
```

We can then put it all together into a single demonstration web-app like so.

```
from flask import Flask, session

app = Flask(__name__)
app.secret_key = 'A0Zr98j/3yX R~XHH!jmN]LWX/,?RT'

@app.route('/')
def index():
    return "Root route for the sessions example"

@app.route('/session/write/<name>/')
def write(name=None):
    session['name'] = name
    return "Wrote %s into 'name' key of session" % name

@app.route('/session/read/')
def read():
    try:
      if(session['name']):
          return str(session['name'])
    except KeyError:
      pass
    return "No session variable set for 'name' key"

@app.route('/session/remove/')
```

[2]Unless the communication has been secured with HTTP but that is another topic

[3]If you are unsure about Python dictionaries or 'dicts' then you should do some background reading on this topic in the Python language documentation

[4]Again, if you are unsure about Python errors & try-execept then you should do background reading on this

```
def remove():
    session.pop('name', None)
    return "Removed key 'name' from session"
```

Notice the 'app.secret_key' line. This is our secret key that is used to secure our session cookie so should really be stored securely, either in a config file or typed in by hand at startup, but never put in the code repository. However for demonstration purposes this is sufficient for now. The key above is sufficient for the lab work but you would generate a unique key for any real deployment and would keep it secret. You can generate a key easily using Python. Start the Python interpreter and use the os.urandom function to generate a new key that you can then copy and paste into your Flask app, e.g.

```
$ uv run Python
...
>>> import os
>>> os.urandom(24)
'\xfd{H\xe5<\x95\xf9\xe3\x96.5\xd1\x010<!\xd5\xa2\xa0\x9fR"\xa1\xa8'
```

2.4 Message Flashing

User feedback is an important consideration when trying to design a good user experience (UX). Message flashing is just one aspect of UX that Flask provides to enable easy user feedback. The scenario is quite simple, when the user does something on one page, which causes another page to be rendered and displayed, then information, a message, can be transmitted from the first page to the second, and displayed, e.g. flashed, to the user. This means that, used with the correct combinations of responses, users can interact with your web app and get feedback about the outcome of actions. A simple example will illustrate this; if you have a sign-up page for new users on which the user enters information then presses a "join" button you can use message flashing to provide a personalised message to the user on the next page that is displayed. For example, after pressing "join" either the sign up page will be redisplayed, because the supplied information is insufficient, or else the login page will be displayed. A flashed message could be displayed in either case, on the sign-up page to indicate what needs to be fixed, or on the log in page indicating that a new account was created and that the user is welcome to log in. What is essentially happening is that a message is recorded at the end of the first request, which is then accessed *only* on the very next request.

Message flashing sounds quite complex and does have a few moving parts, but is really very simple once you have used it once to twice. It is best shown via example, so let's get started with one

```
from flask import Flask, flash, redirect, request, url_for, render_template

app = Flask(__name__)
app.secret_key = 'supersecret'

@app.route('/')
def index():
    return render_template('index.html')

@app.route('/login/')
@app.route('/login/<message>')
def login(message=None):
    if (message != None):
        flash(message)
    else:
        flash(u'A default message')
    return redirect(url_for('index'))
```

Here, we have added a flashed message in one route, using the flash() function of Flask then we use the get_flashed_messages() method in our template to retrieve the flashed message and display it, e.g.

```
<html>
<body>
{% with messages = get_flashed_messages() %}
```

```
  {% if messages %}
    <ul>
    {% for message in messages %}
      <li>{{ message | safe }}</li>
    {% endfor %}
    </ul>
  {% endif %}
{% endwith %}
</body>
</html>
```

All that we have done here is add a flashed messagein the '/login/<message>' route then to display the flashed message when we visit the '/' page. Subsequent visits to the '/' page will not repeat the flashed message, unless of course, we visit the 'login/<message>' page again

3 Logging

We can set our Flask app up to record interesting happenings into a text file so that we have a log to inspect if something bad occurs. This is actually a feature of the Python language rather than a Flask feature, but is an important part of building a real world app. First we need to make some additions to our config file, named 'logging.cfg' and stored in the 'etc' folder:

```
[config]
debug =  True
ip_address = 0.0.0.0
port = 5000
url = http://127.0.0.1:5000
[logging]
name = loggingapp.log
location = var/
level = DEBUG
```

Notice the logging section towards the end which defines the name of the log file 'loggingapp.log' the location, our var/ directory that we just created, and the default log level, the granularity of the logging events to record.

We now need to set up our envionrment to match the config file. We need a subdirectory called 'var' which is at the same level as our 'etc', 'static' and 'templates' directories. We now need to create an empty file in 'var' which has the same name as the log file. This directory will be the location that our new log files will write to. We can do this with touch, e.g.

```
$ mkdir var
$ touch var/loggingapp.log
```

Having set everything up all the configuration details and the basic environment we now need to make use of that set up within our python app. Create a new file called 'loggingapp.py' and enter the following code:

```
import configparser
import logging

from logging.handlers import RotatingFileHandler
from flask import Flask, url_for

app = Flask(__name__)

@app.route('/')
def root():
        this_route = url_for('.root')
        app.logger.info("Logging a test message from "+this_route)
        return "Hello Napier from the configuration testing app (Now with added logging)"

def init(app):
    config = configparser.ConfigParser()
    try:
        config_location = "etc/logging.cfg"
```

```
        config.read(config_location)

        app.config['DEBUG'] = config.get("config", "debug")
        app.config['ip_address'] = config.get("config", "ip_address")
        app.config['port'] = config.get("config", "port")
        app.config['url'] = config.get("config", "url")

        app.config['log_file'] = config.get("logging", "name")
        app.config['log_location'] = config.get("logging", "location")
        app.config['log_level'] = config.get("logging", "level")
    except:
        print("Could not read configs from: ", config_location)


def logs(app):
    log_pathname = app.config['log_location'] + app.config['log_file']
    file_handler = RotatingFileHandler(log_pathname, maxBytes=1024* 1024 * 10 , backupCount=1024)
    file_handler.setLevel( app.config['log_level'] )
    formatter = logging.Formatter("%(levelname)s | %(asctime)s |  %(module)s | %(funcName)s | %(message)s")
    file_handler.setFormatter(formatter)
    app.logger.setLevel( app.config['log_level'] )
    app.logger.addHandler(file_handler)

init(app)
logs(app)
```

The init function is very similar to the one we used earlier in the config example but is now extended to also configure event logging. There are many parameters to fine tune how data is logged so if you want to make changes then you will have to dig into the Python logging documentation but for now, just accept the defaults as they produce files that cope well with lots of data, are easy to read, and which can be automatically processed. All we have had to do to use the logger is to initialise the logging system by calling our configuration modules logs() function. From then on we can actually log messages using the app.logger.warn() and passing in a String. NB. There are also other logging levels such as debug or error that we can use to distinguish the importance of different messages in the logs. Investigate the Python logging documentation to find out more.

Now, if everything is set up correctly, then every time we visit `http://localhost:5000/` a new log file should be added to var/loggingapp.log and we can look at that file to see what has been happeing in our running app. For example, we might see output like the following:

```
INFO | 2015-10-13 17:44:37,368 |  logs | root | Logging a test message from /
INFO | 2015-10-13 17:44:45,104 |  logs | root | Logging a test message from /
INFO | 2015-10-13 17:44:46,325 |  logs | root | Logging a test message from /
```

From this we can see that there were three log messages displayed and that the output is arranged into columns. Getting the output nice and neat like this is the main reason we had to write so much code in our python app and config file, but it is really worth it when you are trying to track down a problem.

It is a good idea to log anything that you think that you might want to have a record of and that you might need to look up in order to bug fix. Unfortunately though there are no rules for what to log and what not to log. Obviously you could log *everything* but this would possibly waste disk space, but not logging enough might mean that you don't have sufficient logs to help you fix problems. However, with experimentation and experience you will develop skills in gauging the right amount of and type of data to log.

3.1 Testing

We can unit test our Python app, to ensure that everything is working correctly. We do this by running a test harness which sets up our Flask app then compares expected outputs to actual outputs, for example, for the following simple web-app (testing.py):

```
from flask import Flask
app = Flask(__name__)

```

```
@app.route('/')
def root():
  return "HELLO NAPIER", 200
```

We can write tests that compare the response returned by a call to the '/' route against what we would expect. For example, is the content correct, is the content type set corretly, is the status value set correctly. Let's see some of these tests now from the testing_test.py file. Notice that testing_test.py imports our source file (testing) as well as the unittest library:

```
1 import unittest
2 import testing
3
4 class TestingTest(unittest.TestCase):
5   def test_root(self):
6     self.app = testing.app.test_client()
7     out = self.app.get('/')
8     assert '200 OK' in out.status
9     assert 'charset=utf-8' in out.content_type
10    assert 'text/html' in out.content_type
```

We have create a class for the test and asserted that the response contents match some of our expectations, e.g. that the response status is '200 OK'. Notice in line 2 that we have also imported our flask app source file so that it is accessible from this test script. To use this we just run it in the shell, e.g.

```
$ uv run python -m testing_test
.
----------------------------------------------------------------------
Ran 1 test in 0.476s

OK
```

We can easily test what would happen if a test was failed by setting things up that way, let's return 404 from our root function, e.g.

```
@app.route('/')
def root():
  return "HELLO NAPIER", 404
```

Now the output from our test suite shoudl be similar to the following:

```
$ uv run python -m testing_test
F
======================================================================
FAIL: test_root (__main__.TestingTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "testing_test.py", line 9, in test_root
    assert '200 OK' in out.status
AssertionError

----------------------------------------------------------------------
Ran 1 test in 0.416s

FAILED (failures=1)
```

We can write as many tests as we need to to ensure that our web-app, or any Python app, runs the way that we expect it to. We do this by writing a new class in our testing script, called a *test harness*, for each test that we want to run. The unit testing framework will then discover each test class and run them, then output the results so that you know, after each edit of your code, whether you accidentally broke anything or, more importantly, changed the behaviour of code that worked previously. Testing is very important and unit testing is just one tool that we have to help us with working with larger bodies of source code. As a code base gets bigger it can sometimes be difficult to tell with a given change has subtly altered the behaviour of something else. Unit testing gives us more confidence that we haven't done so.

3.2 HTML Headers

Remember how we've talked earlier about how the HTTP communications between the client and server have a head part and a body part. Both head and body can contain useful information, both information necessary for HTTP communication, but also additional information to facilitate other communications between client and server related to security and privacy. For example, the client might provide a key in a header field to communicate that the client has already gone through an authentication process. By checking the value of the key the server can determine whether the request is from an authorised user or not.

Some examples of headers:

Content-Type: By setting the "Content-Type" header the server can communicate to the client the type of data being sent. For example, "text/html" to indication HTML content or "application/json" to indicate JSON data.

Cache-Control: Setting the "Cache-Control" and "Expires" headers enables the server to influence how the client caches the response. This can help improve performance by managing the number of requests made to the server.

Access-Control-Allow-Origin: This header is used in cross-origin resource sharing (CORS) to specify which origins, essentially Web domains, are allowed to access the resource. Essentially this means that a server can specify whether Javascript in the browser on other sites can access and manipulate data from this site. As a rule, many sites want to restrict access to their data from other web page as this can often be used to gain access from insecure pages like a phising site, to data on secure pages, like a bank's Web page. However, some sites might specifically be designed to share data, for example, through an API to the JS on the web pages of other sites, and therefore may use CORS to specifically allow other sites to access the resource from other origins.

Location: If you've reorganised your site and are sending redirect status codes[5] then the "Location" header can be used specify the new destination URL that the client should navigate to.

Being able to access the headers set by the client during it's request, or to set headers in your server-generated response can therefore be really useful. Let's take a look at an example. We'll start by setting two headers. In this case setting the "Content-Type" to *plain/text* and the "X-Custom-Header" to *Napier Custom Value*. We do this by adding custom keys and values to the headers dictionar of our response object. We get the response object by call the make_response Flask function[6].

```
from flask import Flask, make_response

app = Flask(__name__)

@app.route('/')
def index():
    response = make_response('Hello, world!')
    response.headers['Content-Type'] = 'text/plain'
    response.headers['X-Custom-Header'] = 'Napier Custom Value'
    return response

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

We can use a similar approach, adding more, or fewer response.headers lines to account for whichever headers we want to set. You can inspect the headers from within your browser's developer tools[7], using cURL, or, depending upon CORS, from withing Javascript.

[5] 301, 302, 307, 308

[6] https://flask.palletsprojects.com/en/stable/api/#flask.Flask.make_response

[7] In the Network tab, refresh the page, select the HTML response, and then headers sub-menu

It might be the case that we want to add custome headers to every response in our site. In which case we can use the after_request decorator[8] to write the headers after every call to our Flask app.

```
from flask import Flask, make_response

app = Flask(__name__)

@app.route('/')
def index():
    return 'Hello, world!'

@app.after_request
def set_response_headers(response):
    response.headers['Content-Type'] = 'text/plain'
    response.headers['X-Custom-Header'] = 'Custom Napier Value'
    return response
```

As well as setting headers on the server to return to the client, we can also inspect what the client has sent to the server. For example, to find out how your client Web browser is identifying itself in requests to your Flask app, you can use the following:

```
from flask import Flask, request

app = Flask(__name__)

@app.route('/')
def index():
    user_agent = request.headers.get('User-Agent')
    return f"User-Agent: {user_agent}"
```

Note that this code uses a feature of Python that you might not have come across yet, f-strings[9]. It's really just an easy way to embed variables into a string. Note that you can access the headers sent in the request, but these are immutable, that is, they cannot be altered and will cause an error if you try to do so.

4 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. Add some custom headers to one of your earlier Flask apps then investigate accessing them from cURL and inspecting them via your browser's developer tools.
2. Create a simple maths quiz flask app that uses a form to post the players answer. Use message flashing to indicate whether the submitted answer is correct or not.
3. For your maths quiz, use a session to store the users current score

If you're still not confident with creating a new Flask web-app from scratch[10] 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.

[8] https://flask.palletsprojects.com/en/stable/api/#flask.Flask.after_request
[9] https://www.geeksforgeeks.org/python/formatted-string-literals-f-strings-python/
[10] you should be able to do the entire pprocess from the first lab topic in less than five minutes by now.

5 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.