Edinburgh Napier University

SET09103 - Advanced Web Technologies

Lab 03 - Methods & Status Codes

Dr Simon Wells

1 Overview

GOAL: Our aim in this practical is to elaborate on our simple Flask web-app up from before to add some different features from HTTP. This means the following:

1. Utilising HTTP methods in our interactions with our routes.
2. Returning different status codes.
3. Handling HTML form data.
4. URL variables & URL parameters.
5. Working with requests objects.
6. Working with response objects.

Again, the 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 thinkgs 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

The last topic should have got us to the point where we can build a simple 'greetings' web app that extended the basic 'Hello Napier' web app. It's important at this point that you are comfortable with the basic tooling of uv, Python, virtual environments, and Python libraries. You must also be comfortable with using basic Flask functionalities to create a simple working web-app that returns pages from a range of addresses.

We can now start to expand our web-app skills using Python-Flask. We'll start by looking at HTTP methods, then take a slight diversion into how we can build apps using multiple methods per web address and respond with different pages for each. We'll exploit this to show how we can create a page containing a form then process the form and return a results page using a sequence of methods. We'll then take an even larger diversion into URL variables and URL parameters. This will pretty much finish our sub-topic on Web addresses using Flask because we'll have all of the parts in place for handling the information that could be put into the address bar. After that we'll look at the response object, how we can inspect it, and introduce some additional tools to support that process. Finally we'll introduce HTTP status codes.

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.

So far we've learned to create different Web pages and public routes so that we can use the browser to retrieve those pages, Basically, we execute a different function depending upon which URL the browser requested. Our pages still look fairly rudimentary[2]. In this topic we will look at the requests, and their counterparts, the responses, in more detail. Because requests can include more than just a simple HTTP GET we shall look at how to handle different HTTP methods, such as GET, POST, PUT, & DELETE. Then, after receiving and handling request data, we will look at the responses that we can return to the client.

2.2 Requests

When your browser connects to a URL it is making a **r**equest. So when your browser requests a route from your Flask app, it sends an HTTP request message to your web-app. This request is converted into a Python Flask object that we can can access, inspect, and whose data we can reuse. For example, a request might carry a payload such as a document associated with a POST request and we will likely need to retrieve that payload document from the request in order to process the data and return an appropriate response. This is particularly important if we want to incorporate HTML forms into our Flask web-apps. When your browser sends the data from a form to the server, the browser creates an HTTP request message and the form's data is encoded and incorporated into the request. Again, when the request arrives, it is decoded and a Python Flask object is created to hold the incoming request information.

You can, for development, debugging, and educational purposes, investigate the request object by printing the request.method, request.path, and request.form attributes to retrieve data about the actual request, e.g. within a route in your Flask app you cab get more information about the context of the request so that your web app can provide a more appropriate response. Recall that we discussed how HTTP is a protocol for communication? Well an important part of good commmunication is responding appropriately to messages. HTTP gives us the power to work out just what the client wants and to use all of the request information to tailor a specific response within the remit of the functionality of the web-app that the route is a part of.

```
1 from flask import Flask
2 app = Flask(__name__)
3
4 @app.route("/")
5 def hello():
6     print (request.method, request.path, request.form)
7     return "<p>Hello World!</p>"
```

Note that if some of these attributes of the request object don't contain any information, for example, if there isn't any form data, then you might get *None* as the result.

2.3 HTTP Methods

HTTP uses various methods or *verbs*[3] to move data around. The most commonly used method, and the one we have used exclusively until now is the default GET method. There are however many HTTP verbs[4] as defined in the HTTP standard.

Look at the output from the Python Flask development server and you should see lines similar to the following:

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

[2] at least they will unless either, you read ahead, or you wait until next week when we consider a few methods for working with HTML documents and templates

[3] because they always correspond to doing words, you're requesting some action

[4] see `https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Methods`

```
10.0.2.2 - - [27/Sep/2015 18:59:51] "GET / HTTP/1.1" 200 -
10.0.2.2 - - [27/Sep/2015 18:59:58] "GET /hello HTTP/1.1" 200 -
10.0.2.2 - - [27/Sep/2015 19:00:05] "GET /goodbye HTTP/1.1" 200 -
```

Notice the part of each line that says GET? This is because we have been interested only in retrieving information using the *de facto* default HTTP method.

HTTP specifies a range of methods for requesting web resources and these methods are oftern referred to as HTTP Verbs. By default a client usually makes GET requests and most web resources will respond to a GET request, however a resource, identified by a route, can respond differently to different verbs. So, for example, we can make a GET request to retrieve a resource or a POST request to send information to the resource. We can then write code to respond to differnt requests in different ways.

As we said, a Flask route will accept GET requests by default, and Flask also supports HEAD requests automatically when GET is present, but we can also add support for other verbs to the route decorator method, e.g. to specify that a route can accept both GET and POST requests we would use the following decorator:

```
1 @app.route('/account', methods=['GET', 'POST'])
```

Within the method associated with that decorator we could then use an *if...else* block to execute different code, e.g.

```
1  from flask import Flask, request
2  app = Flask(__name__)
3
4  @app.route("/")
5  def root():
6    return "The default, 'root' route"
7
8  @app.route("/account/", methods=['GET', 'POST'])
9  def account():
10   if request.method == 'POST':
11     return "POST'ed to /account root\n"
12   else:
13     return "GET /account root"
```

To test this we can browse to `localhost:5000/account` and we should see the result of GET'ting the request.

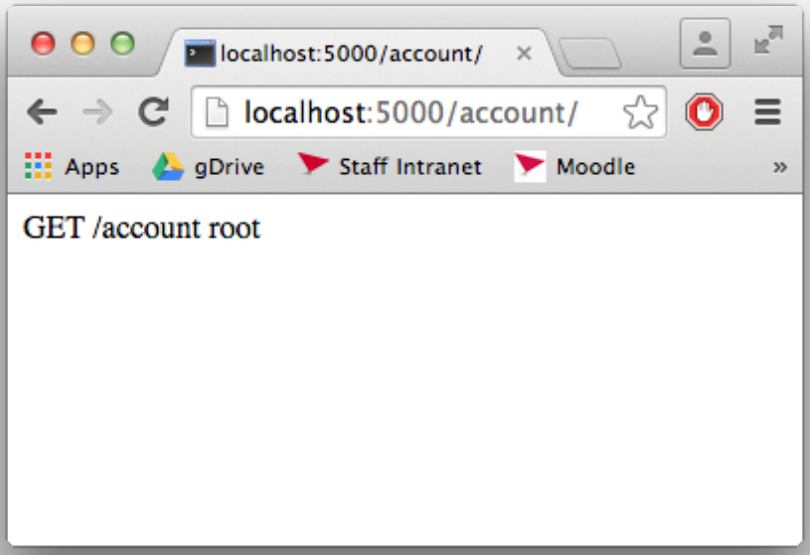

Figure 1: Result from GET'ting our account/ route

To test the POST call is slightly more involved for now. As we haven't implemented any HTML yet that is capable of calling this root using a POST request we need to mock one up. We can use the cURL command line tool for that[5]. Open a new terminal, but leave your existing one with your flask app running, and then use the following command

```
$ curl -i -X POST localhost:5000/account/
```

obviously replacing 5000 with the port that your API is running on if you've deployed elsewhere (like 80 or 8080). This will give output similar to the following[6]:

```
HTTP/1.0 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 24
Server: Werkzeug/0.10.4 Python/3.12.15
Date: Sun, 04 Oct 2025 12:51:12 GMT

POST'ed to /account root
$
```

We can find out what HTTP method was used by checking the request method (as we saw in section 2.3). Data transmitted in a POST or PUT request can then be accessed using the form attribute of the request object.

2.4 Requests & Request Form Data

Now let's look at an example that displays a form when we connect to the URL using GET then display a different page that uses data from the form when we submit a POST request by pressing the form's button. This is interesting because it essentially means that the account route is providing two different Web pages depending upon how it is interacted with and the only difference is the use of an HTTP verb. If we access the route with one verb we get one page and if we access the route with anbother verb we get a different page. To make the example cohesive though, the two pages are related. The first page, returned when we access the route from a browser, returns a simple HTML form into which you can enter a name. The submit button causes the next call to

[5]cURL is installed by default on most Linux desktop installs, as well as MacOS, and Windows 10 and 11

[6]minor changes in version numbers and date/time is to be expected, and perhaps more significant differences between platforms, but the general format is the same

account to issue an HTTP POST request, instead of the default GET request[7] It is worth taking some time to trace through the account method in detail. This pattern of testing some incoming data from the request, then using an if-else clause to select between different responses

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

@app.route("/account/", methods=['POST','GET'])
def account():
  if request.method == 'POST':
    print (request.form)
    name = request.form['name']
    return "Hello %s" % name
  else:
    page ='''
    <html><body>
      <form action="" method="post" name="form">
        <label for="name">Name:</label>
        <input type="text" name="name" id="name"/>
        <input type="submit" name="submit" id="submit"/>
      </form>
      </body><html>'''

    return page
```

This should yield something similar to the following when we visit `http://localhost:5000/account/` in the browser:

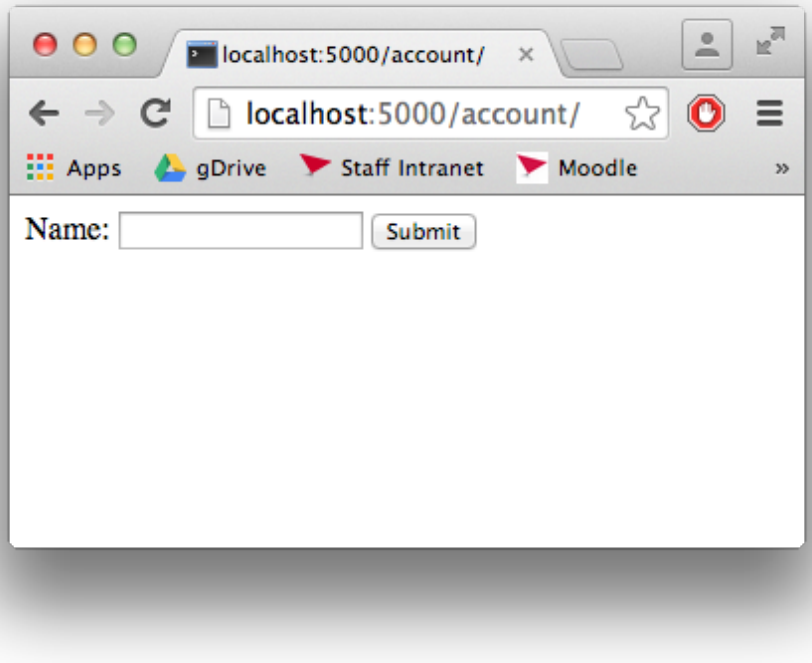

Figure 2: Our basic HTML form when GET'ting the account route

It is worth noting how we have used a Python multiline string, which starts with "' and ends with "' to build up an html page completely within out Python function. When we enter data into the input box then click the *submit* button we should get different page displayed as a result of the form POST'ing:

[7]By convention, browsers only issue GET requests except when sending form data, in which case they issue POST requests. If you want to use other verbs then you need to use a tool other than a Web browser as your client.

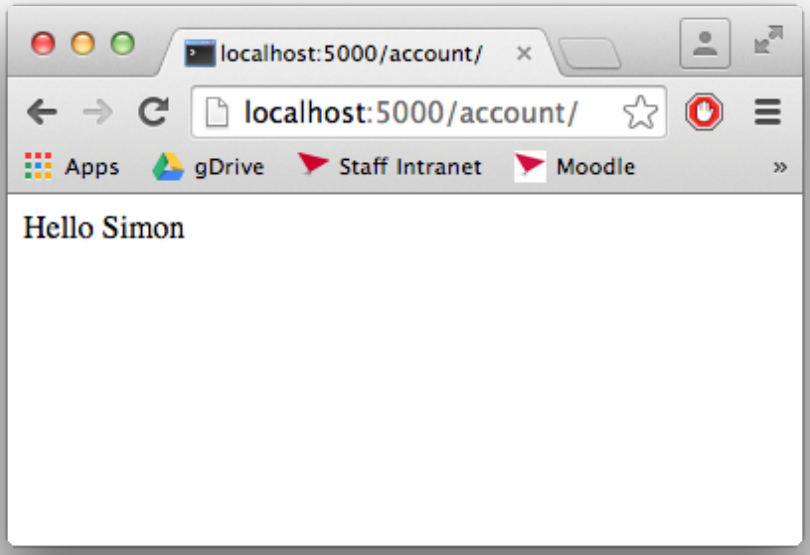

Figure 3: Page displayed after POST'ing the form

We will return to using POST, PUT, and other verbs in later topics, in particular, when we look at designing and building APIs that consume and return JSON and XML documents.

2.5 URL Variables

Now let's take a very slight, but informative and useful, detour to consider how we can send data to the server using the address. That is, using URL variables and URL parameters. Url variables are a way to create a Web address where elements of that address are *variable*. For example, . By contrast, URL parameters, which we'll see next, are a way to add key=value pairs of labelled inforamtion to the end of a Web address and to then access the values of those parameters from within our code.

We can construct URLs that have variables within them. For example, if we wanted a URL route that enabled us to retrieve a user's details by name then we might want a URL of the following pattern */account/<username>* where <username> is replaced by an actual name so the actual address would change depending upon whether we wanted */account/simon* or */account/thomas*. We can achieve this using URL variables, e.g.

```
1 from flask import Flask
2 app = Flask(__name__)
3
4 @app.route("/hello/<name>")
5 def hello(name):
6   return "Hello %s" % name
```

We can now call the url, e.g. using the name 'simon' in the following `http://set09103.napier.ac.uk:5000/hello/simon` and we would get the following output:

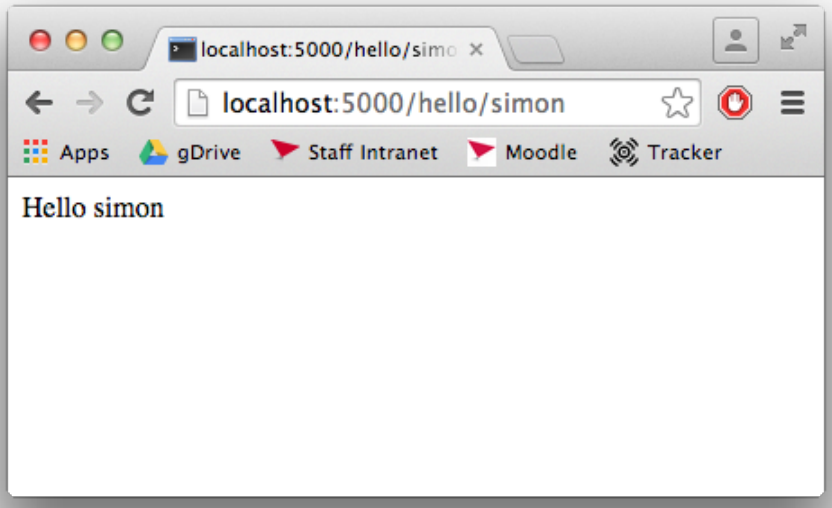

Figure 4: Using URL variables

By default URL variables are strings but we ca also specify other variable types such as int and floats, for example,

```
1 from flask import Flask
2 app = Flask(__name__)
3
4 @app.route("/add/<int:first>/<int:second>")
5 def add(first, second):
6   return str(first+second)
```

With the following result:

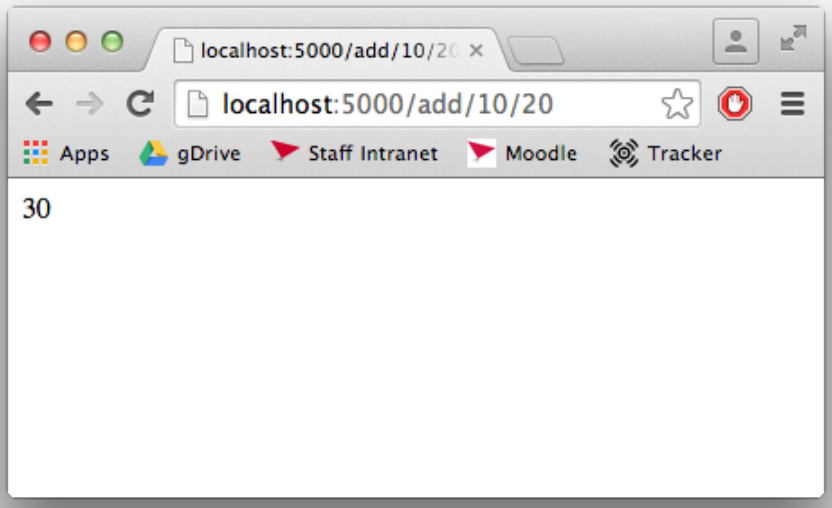

Figure 5: Output from using specific URL variable types

2.6 URL Parameters

Rather than construct and send a document to the server or use a POST'ed form, we will often want to send small amount of non-secure data to the server encoded within the URL. This is straightforward in Flask.

Parameters can be encoded in the URL when a client make a request. Flask can retrieve these parameters and use them by using the args attribute of the request object, e.g. for a URL that incorporates *?key=value* parameters similar to the following:

```
/update?colour=green
```

Then we can access the corresponding keys like so:

```
1 searchterm = request.args.get('key', '')
```

The value for key is then retrieved from the URL and stored in 'searchterm'. If no such key is supplied then the value in the second pair of quotes is used as a default instead but in the example above we have just used an empty string.

Now let's look at a simple example, which you can use to send your name as a URL encoded parameter, and have a route accept and process it we can do the following:

```
1  from flask import Flask, request
2  app = Flask(__name__)
3
4  @app.route("/hello/")
5  def hello():
6    name = request.args.get('name', '')
7    if name == '':
8      return "no param supplied"
9    else:
10     return "Hello %s" % name
```

When we run this without supplying a parameter, e.g `http://set09103.napier.ac.uk:5000/hello/` then we get this output:

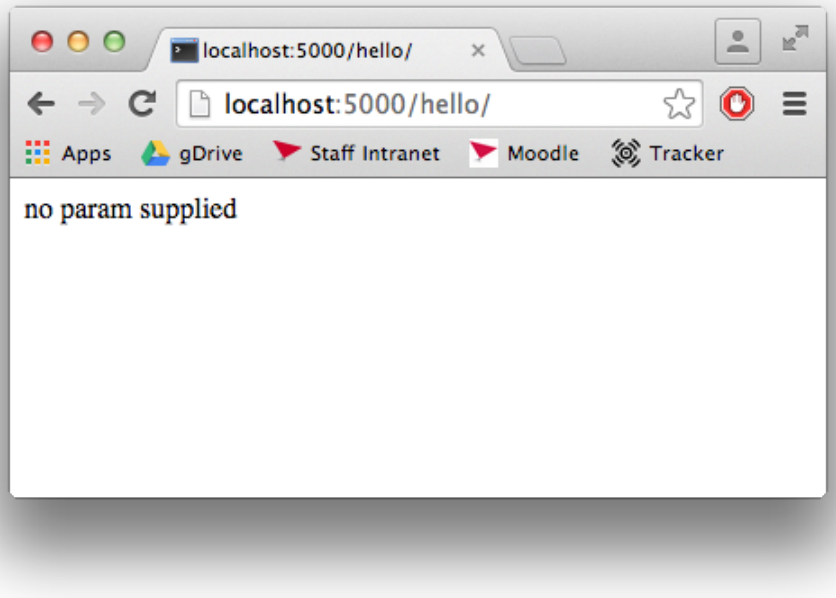

Figure 6: Output with no URL parameter

When we supply a parameter, e.g. `http://set09103.napier.ac.uk:5000/hello/?name=simon` then we get this output:

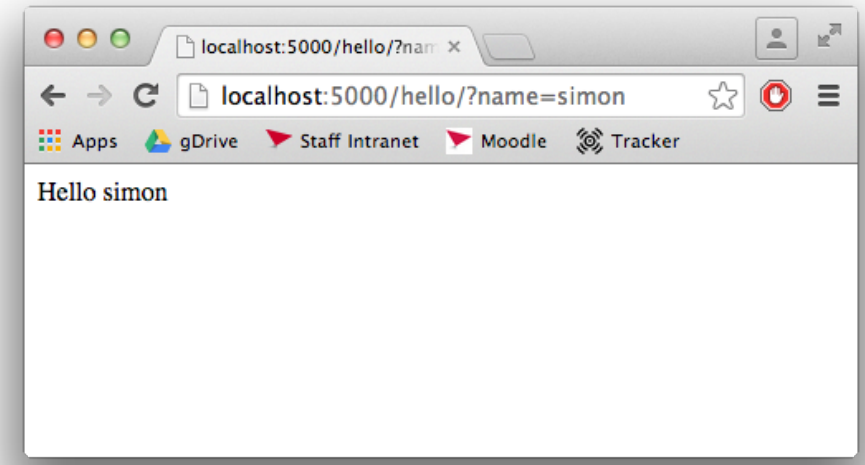

Figure 7: Output when supplying a ?name=simon URL parameter

3 Responses

When we return a value from a function it is automatically turned into a valid HTML response object. This is some magic that Flask does for us. This object is serialised into an HTTP response object that the client Web Browser can understand. If the value is a String then it is used as the body of the response which is why when we just do something like this:

```
1 from flask import Flask
2 app = Flask(__name__)
3
4 @app.route("/")
5 def root():
6     return "Hello Napier"
```

then our browser displays the String that the function returned. A 200/OK HTTP status code is also returned by default and the mimetype is set to text/html. We will return to this topic later when we look at API building and consider setting and returning custom headers. In the next topic we'll also see how different things, other than strings, like HTML files or template files, can be returned instead of just strings.

'

4 Inspecting Requests & Responses

In the lectures we've been considering HTTP as a protocol and have focussed on the things that makes HTTP communication into a language for interactions between the HTTP client and the HTTP server. This involves not only requests and responses but also supplementary aspects of the communication such as HTTP verbs and status codes.

In order to really get a grip on this it can be useful to see what is actually being sent in a real request, and what is being returned in a real response. We're also now at the point when we can manipulate all of those aspcets of HTTP using FLask, so again, it can be usful to see what is actually happening in terms of changes to our requests and responses. We've seen earlier how we can use cURL to acces and test our API, by crafting requests, but we can also use it to inspect both the requests and the response.

By using the -v argument of cURL, where v stands for "verbose", we can get cURL to print more information about what it is sending and receiving. Let's try that now.

```
$ curl -v http://0.0.0.0:5000/
*   Trying 0.0.0.0...
* TCP_NODELAY set
* Connected to 0.0.0.0 (127.0.0.1) port 5000 (#0)
> GET / HTTP/1.1
> Host: 0.0.0.0:5000
> User-Agent: curl/7.64.1
> Accept: */*
>
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Content-Type: text/html; charset=utf-8
< Content-Length: 12
< Server: Werkzeug/2.0.1 Python/3.8.2
< Date: Mon, 27 Sep 2021 12:42:12 GMT
<
* Closing connection 0
Hello World!~
$
```

In this example we've used cURL to access the default route from our 'hello world' Flask app from Section **??** which is running on localhost port 5000 (`http://0.0.0.0.:5000`). There are two parts to our output from cURL, the first is the request that is being sent, and the second is the response that we get. Notice how we get a lot more information by doing this than if we just accessed `http://0.0.0.0.:5000` via our browser. If we did that then all we'd see is "Hello World!" in our browser window. This way we get to see information about the HTTP version, the content encoding, the server software[8].

Note that we can also use the developer tools built into Chrome to inspect the request and response headers for any given site we navigate to. To do this navigate to the page you want to inspect the request for, then open the developer tools and select the 'Network' tab and refresh the page. Now select the URL in the list on the left hand side of the page then the 'Headers' tab on the right hand side to have the headers displayed for you. This separates the headers into the request headers and the response headers as well as providing some general information about the communication as illustrated in Figure 8.

[8] Werkzeug is a library that underpins Flask and is one of the other modules installed when you pip install Flask.

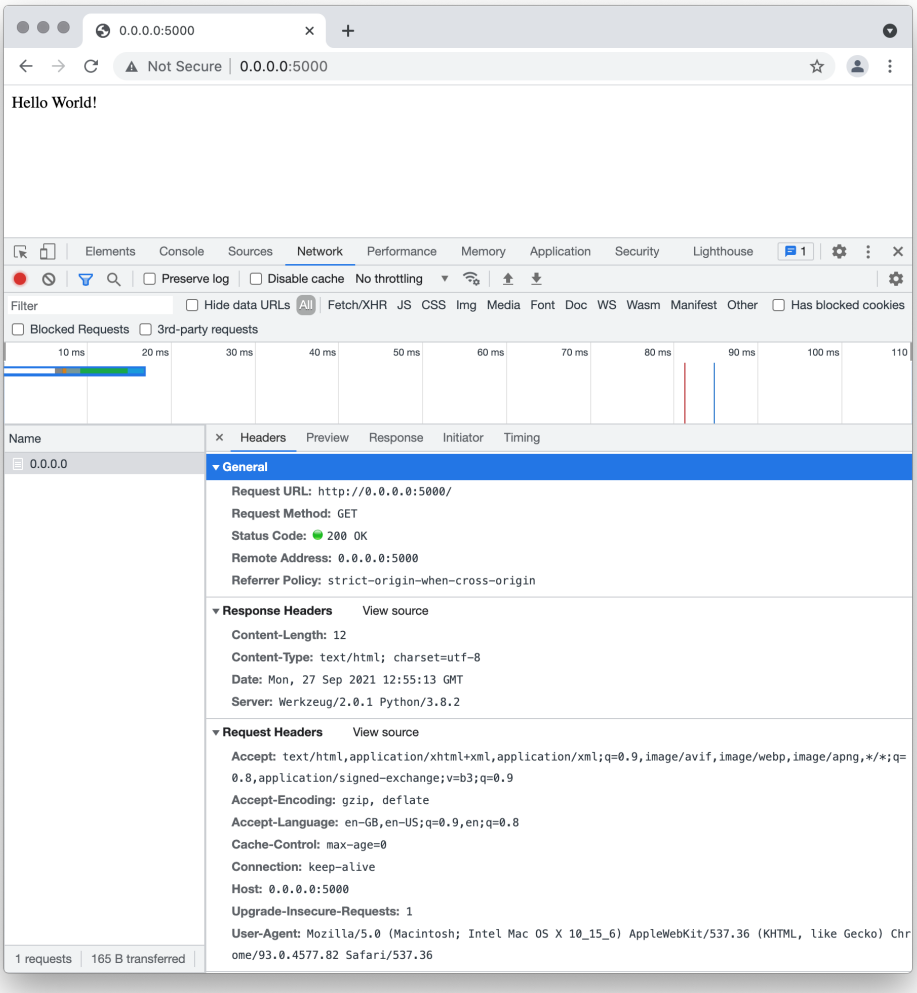

Figure 8: Inspecting request and response headers using the Chrome Developer Tools

4.1 HTTP Status Codes

We'll finish with something really simple. We can arbitrarily add an HTTP status code to a response by appending it to the return string. For example,

```
1 from flask import Flask
2 app = Flask(__name__)
3
4 @app.route("/")
5 def root():
6     return "Hello Napier", 200
```

won't actually give us a different response to before, as a 200 OK code is automatically appended, but does illustrate the pattern for specifying a status code to return. We could instead append 418 to the return line. Take a moment to look up what the 418 status code indicates as well as to explore the range of standardised codes available[9].

```
1 from flask import Flask
2 app = Flask(__name__)
3
4 @app.route("/")
5 def root():
6     return "Hello Napier", 418
```

[9] https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Status

Take a moment to inspect the results from calling each of the previous two examples using cURL and using the browser developer tools.

We shall return to HTTP status codes, and HTTP methods for that matter, later when we consider the design of data APIs and the role that both good selection of methods and good selection of status codes play when creating an API that self documents and communicates clearly.

5 Exercises

Taking any of the Flask apps that you've created so far, including in earlier labs, as a starting point:

1. Create a dice web app that will return random dice rolls when called. Use the GET verb to retrieve a simple form that enables you to specify how many dice to roll. Then use the POST'ed form data to set the number of dice to roll. Calculate the result and display it in the resulting page.
2. Create a new version of your dice web app that uses URL parameters to set the number of dice to roll then displays the result.
3. Create a web app that uses a colour passed in as a URL parameter to determine which colour to set the web page background to, e.g. For the following route: `localhost:5000/?colour=green` the background should be set to green. Consider how you might get this to work for all of the colours that a browser recognises by name[10].

It's important to get to grips with the process of creating a new Flask web-apps so don't just edit and re-use your existing ones. 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.

6 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.

[10] as defined in the HTML standard.