Edinburgh Napier University

SET09103 - Advanced Web Technologies

Lab 02 - Debug, Errors, Routing, & Redirects

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. This means the following:

1. Run in Debug mode and understand it's utility.
2. Start to build some insight into understanding and dealing with errors.
3. Building simple multi-page sites using Python Flask

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.

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 'Hello Napier' web app using Python and Python-Flask supported by the uv tooling. We can now start to expand our web-app skills using Python-Flask. We'll start by looking at the debug mode which enables us to tell the Flask development server to do hot reloads of updated code whenever we save our edited python web-app file. This saves us lots of time as we begin to make lots of changes to our web-app. Additionally, we should expect to to see lots of errors. After all we are working with lots of new tools. So we shall look at Flask errors and how to deal with them. Then we will look at routing requests from web browsers to different URLs, basically how to build a tree of web addresses that each fire off a different Python function when the browser tries to access them.

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.

2.2 Debug Mode

Debug mode was introduced at the very end of the previous lab sheet but it is quite an important concept so we'll expand on it this week. There are a few things that it helps us with:

1. Hot-reloading code
2. Giving us useful Python stack-traces within the browser

Basically it is really useful to save you time and effort during development. But you *really* need to turn it off if you deploy your web-app publicly.

Let's start by considering the Flask debug mode in a little more detail. Recall that our 'Hello Napier' app looks something like this:

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

When we run this Python flask app using the flask run command in the terminal, e.g.

```
$ uv run flask --app hello run
 * Serving Flask app 'hello'
 * Debug mode: off
 WARNING: This is a development server. Do not use it in a production deployment. Use a
     production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Running on http://146.176.251.30:5000
```

Python calls the run function of the Flask app object and executes it. By default run() takes no arguments. Both the app object and the run function have a number of other features that we can use[1].

Whilst developing a new web-app one of the most useful things we can take advantage of is the debug mode. There are several ways to turn this on, but the best is to use the debug option when we start the development server. This essentially means adding '–debug' to the end of our invocation to Flask, e.g.

```
$ uv run flask --app hello run --debug
```

When we use this command we get slightly different output from before (because we are now running in debug mode instead of normal mode):

```
$ uv run flask --app hello run --debug
* Serving Flask app 'hello'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a
    production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5000
* Running on http://146.176.251.30:5000
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 267-812-569
```

Two important features of the debug mode are:

1. Causing the development server to be automatically restarted each time we change our code, e.g. each time we save our file after editing it in our editor.

2. Printing out debug information and a Python stack trace in the browser so that we can work out what went wrong. Note that the stack trace is also output to the terminal in which your Flask app was invoked, but this is mixed with the output from the server and can make things more complicated to understand.

So from now on, during development, remember to add '–debug' to the end of your invocation to get Flask running. It will really save you time because you won't have to restart the server after every edit and save of your source code. Every little thing that you can do that saves you effort, or clicks, or window management, will often save you time.

[1] We'll consider some as we move through the module and practical tasks but take a look at the Flask documentation if you want to know more right now: `https://flask.palletsprojects.com/en/stable/`

2.3 Errors

If we write code that causes a problem within FLask then we will get an error. You will create Flask apps with errors in them, but we can also force an error using the abort() function. For example:

```
1 from flask import Flask
2 app = Flask(__name__)
3
4 @app.route("/")
5 def error():
6     doerror()
```

All this simple Flask app does it create a single route which, when accessed, causes a Python error. We've done this specifically to show what it can be like when an error happens. If we are running in normal mode, without the debug mode running, then we will see something like this:

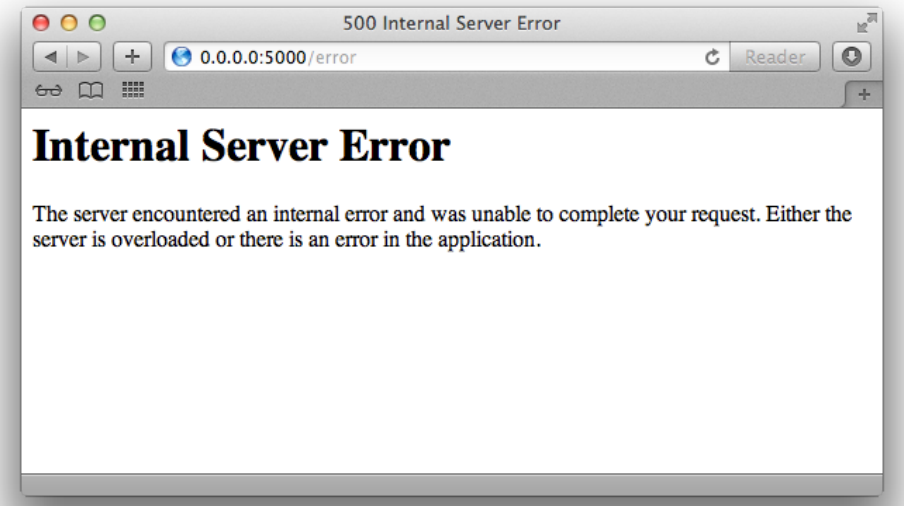

Figure 1: Flask internal server error

What does it tell us? Not much more than that there has been an error. It doesn't tell us the kind of error, or give us any detail about the specific situation that lead to the error occurring, just informs us politely that the error has occurred.

If we enable the debug mode then instead of the generic error, we will get a Python stack trace displayed in the browser (and printed on the output in the terminal where you ran the web-app). A stack trace is really useful because, in many cases, it will give you specific information about exactly where in the code the error occurred. In which case you can go to that location in your source code, fix the error, and re-try your web-app. In some cases, the error might have arisen due to something else going awry slightly earlier, so you sometimes have to read up through the stack-trace to see if the source of the error is made evident. Because you will encounter a lot of errors, you will start to develop a feel for reading the stack-trace and locating the source of the problem. The whole stack trace unwinds everything from the error, through your code, and back into the libraries used by your code, like Flask, and eventually to the Python language itself. Most often you will find that the problem lies in your code rather than in the libraries or Python language itself, but that does happen on occasion, so don't discount it entirely.

Here is an example stack trace but note that the stack traces you will see may well differ greatly depending upon the type of error and the details of what went wrong. As a result this is merely indicative, to give you an idea of what to expect:

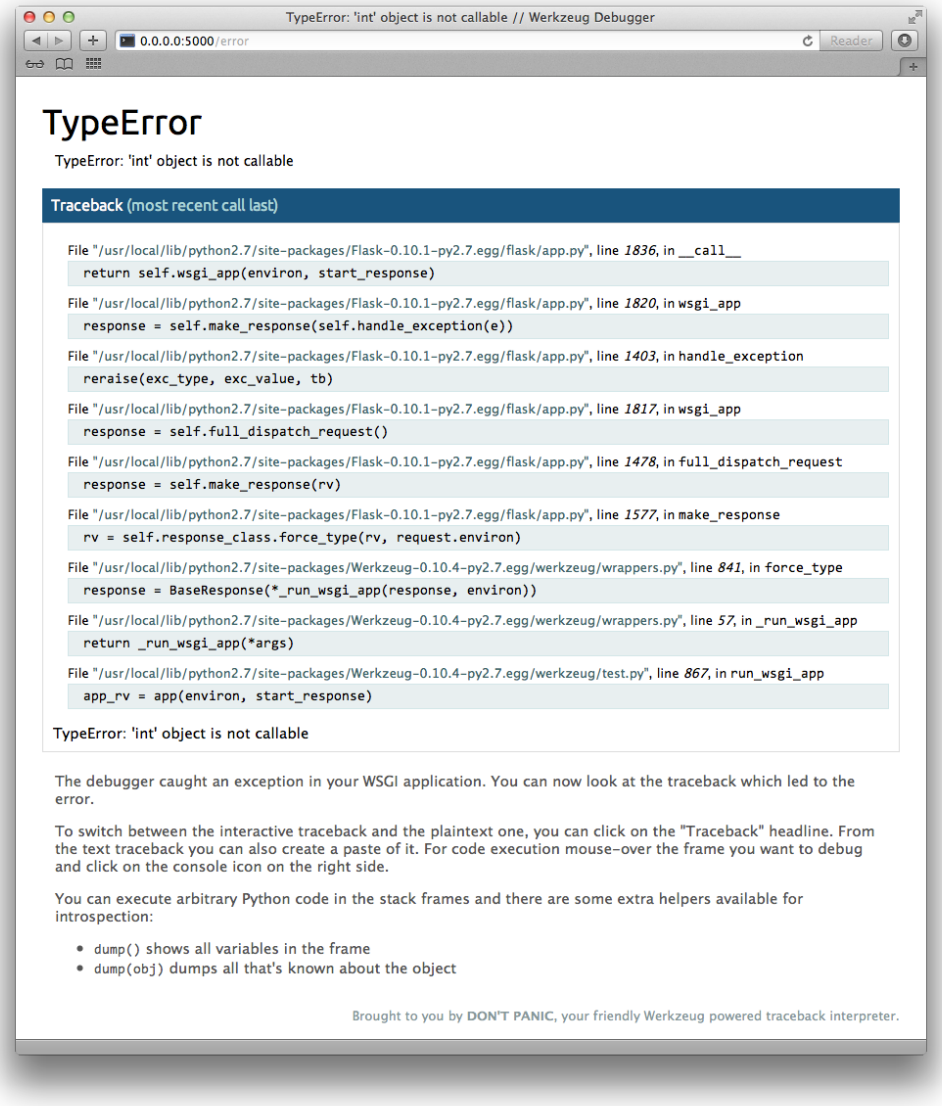

Figure 2: An error stack trace example

2.4 Routing

Routing is what enables us to build sensible URLs and Web-addresses for the pages of our web-app. Most Web-sites have multiple pages, each of which is identified by a URL. In Flask, each page is represented by a Python function. That is, a line beginning with **def**. In fact we've already seen a line beginngin with def in our hello world example. When that function was called it returned a string containing some text enclosed in HTML tags. This string was our page. Don't worry, later we'll see how to put our HTML into other files and then to do interesting things with those files, but for now, we're concentrating on the idea that each page is represented by a function. Notice again in the hello world example that the line immmediately preceding the function line starts with **@**, e.g. **@app.route("/")**. In Python this is a *decorator* and is used specifically by Flask as a way to match a URL to a function.

So to create a page in our Flask app we need three lines:

1. A decorator that defines the Web address (URL).
2. A function definition that defines the code to be run when Flask matches the URL asked for in the request to a URL that is defined in a decorator function.
3. A function body that returns the required data. In our case the required data is a string containing some HTML tags and content.

Later we will elaborate on what the third element, the function body, can contain and also look at other ways to represent the content of our pages instead of just returning a string, but for now, this is the simplest way to get started.

When creating your own Web-apps you should consider very carefully the design of the address hierarchy for your web-app. This hierarchy, the collection of URLs is one part of the API for your Web-app. I think it is almost as imporant a consideration as the design of the content of your actual web-apps pages. A good URL address hierarchy can make the organisation and relationships between your pages really clear and predictable which can lead to a much better user experience.

In Flask, Web addresses or URLs are called *routes*. To add more routes to your web-app you use the app.route decorator as mentioned above. You write a new Python function then add a decorator for it to make the function into a route. For example, in the following web-app, saved in root.py we have 3 routes:

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

@app.route("/")
def root():
  return "The default, 'root' route"

@app.route("/hello/")
def hello():
  return "Hello Napier!!! :D"

@app.route("/goodbye/")
def goodbye():
  return "Goodbye cruel world :("
```

We have the 'root', 'hello', and 'goodbye' routes. Each returns a different, plain text message. We can run this using **uv run flask –app root run –debug**. Notice that now we are creating different web-apps, with different file names, e.g. root.py, we need to tell flask the name of the file to run so it doesn't get confused.

2.5 Redirects

You can redirect a user from one URL endpoint to another quite easily by using the redirect() function, for example, if we were building an app that required a user to be logged in and we detected that the user wasn't actually logged in then we could redirect the user to a login page instead of the page that they requested. This pattern happens a lot with web-sites that have user accounts where different information is returned depending upon the users log-in status, e.g.

```
from flask import Flask, redirect, url_for
app = Flask(__name__)

@app.route("/private")
def private():
  # Test for user logged in failed
  # so redirect to login URL
  return redirect(url_for('login'))

@app.route('/login')
def login():
  return "Now we would get username & password"
```

Note that whilst this runs, it isn't an entire web-app with log-in, just some indicative code for how a redirect could work. To complete the scenario above we would need code to add code to actually check whether the request came from a logged in user and the login page would also need to accept and check any supplied credentials. We will actually look at that type of functionality later, but for now it gives you the idea.

3 Exercises

Taking any of the Flask apps that you've created so far as a starting point:

1. Create a Flask app that incorporates a set of simple web pages to describe yourself and your career at Napier. If that sounds boring then you can choose another topic, such as a hobby, or something that interests you. The main point is to start to consider how the pages relate to each other, how you would navigate between them, what content you would put in them, etc. You should include links to external resources as well as links between your own pages. An important part of this exercise is to *design* your URL hierarchy, then to implement that in your set of pages. For the moment we will continue to place our HTML tags and content into our Python code, but over the next few lab sessions we'll see some other ways to incorporate HTML, as well as CSS and JS so we can make our sites as functional as necessary. The focus of this exercise to relating the HTML content of the page, returned by the Python function, and the route information that relates the page to the URL address in the browser.

2. Create a Flask app that has three routes, 'root', 'ten' and 'ninety'. Implement a redirect so that when the root route is accessed it will randomly redirect to 'ten' 10% of the time and to 'ninety' 90% of the time. You can put whichever content you like into each of the ten and ninety pages. You ight need to look up some Python functionality for generating random numbers[2]

3. Find the CIA world fact website. You might need to use a search engine to do so. The factbook contains information about various countries around the world. Have a look at the kind of information available and design a simple set of pages within a Flask app, that recreates some of the factbook's content. One approach might be to start with an index or "home" page, that introduces things and lists the countries that you will cover. For each country you can then create a "page" covering that country. Your home page can then link to the country specific page. Consider how the information on the page is organised and also how it can be interlinked, either within a page, or between pages. These links give you different ways of navigating through your site and are the essence of *hypertext*. Try to come up with a strategy for how you use links within your site so that navigation is consistent and predictable.

4. Create a Flask Web-app that implements a "choose your own adventure" style game. The home-page should introduce the first part of the story and include links, framed as choices, that take you to other pages with additional parts of the story and further choices, and so on. Eventually, your reader's choices should take them to a page that ends the story, with them as a winner or hero perhaps, or otherwise, dying horribly[3]. The concluding pager

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.

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/`

If you aren't yet 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`

[2] *Hint:* You might need to amke use of the **random** Python library.
[3] which was the main outcome in the original choose your own adventure game books