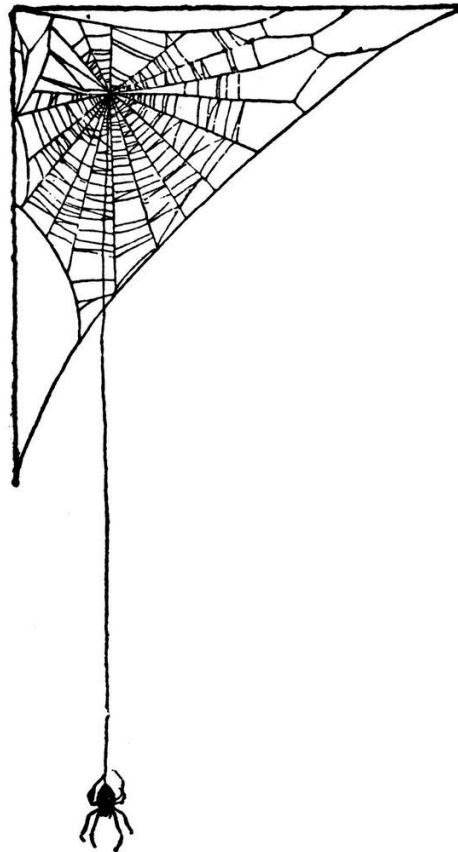


# Web Technologies: Server-Side

COVERING DIVERSE TOPICS RELATED TO  
DYNAMIC SERVER SIDE WEB AND SERVICE  
DEVELOPMENT WITH ATTENTION PAID TO  
THE FUTURE OF THE WEB

BY  
DR SIMON WELLS



SESQUIPEDALIA VERBA PUBLISHING LTD

# Contents

	Page
<b>I Admin</b>	<b>1</b>
1 Introduction	2
2 Overview	4
<b>II Labs &amp; Practical Work</b>	<b>6</b>
<b>3 Learning Environment Part #1</b>	<b>7</b>
3.1 Linux Development Server . . . . .	9
3.2 Alternative Learning Environments (optional) . . . . .	10
3.3 SSH . . . . .	11
3.4 Basic Linux Usage . . . . .	16
3.5 Vim . . . . .	20
3.6 Git . . . . .	22
3.7 Wrapping Up . . . . .	24
<b>4 Learning Environment Part #2</b>	<b>25</b>
4.1 Python . . . . .	26
4.2 Python-Flask . . . . .	28
4.2.1 Installation . . . . .	29
4.3 Python Flask “Hello Napier” . . . . .	31
4.4 Reading & Resources . . . . .	33
4.5 Wrapping Up . . . . .	33
<b>5 Python Flask: Debug Mode, Errors, Routing, &amp; Static Files</b>	<b>35</b>
5.1 Flask Debug Mode . . . . .	35
5.2 Flask Routing . . . . .	38
5.3 Flask Redirects & Errors . . . . .	39
5.4 Flask Static Files . . . . .	42
<b>6 Python Flask: Requests &amp; Responses</b>	<b>46</b>
6.1 Requests . . . . .	46
6.1.1 HTTP Methods . . . . .	46
6.1.2 Request & Request Form Data . . . . .	49
6.1.3 URL Variables . . . . .	51

6.1.4	URL Parameters . . . . .	53
6.1.5	Uploading Files . . . . .	55
6.2	Responses . . . . .	57
6.3	Inspecting Requests & Responses . . . . .	57
<b>7</b>	<b>HTML Templates using Jinja2</b>	<b>60</b>
7.0.1	Templates & Tags . . . . .	60
7.0.2	Templates with Conditional Arguments . . . . .	62
7.0.3	Templates & Collections . . . . .	64
7.0.4	Template Inheritance . . . . .	65
<b>8</b>	<b>Flask: Configs, Sessions, Message Flashing, Logging, &amp; Testing</b>	<b>70</b>
8.1	Configuration & Config Files . . . . .	70
8.2	Sessions . . . . .	71
8.3	Message Flashing . . . . .	73
8.4	Logging . . . . .	74
8.5	Testing . . . . .	77
<b>9</b>	<b>Using Bootstrap to Add Style</b>	<b>79</b>
<b>10</b>	<b>Data Storage</b>	<b>86</b>
10.1	Brief Introduction to SQLite3 . . . . .	86
10.2	Using SQLite3 with Flask . . . . .	89
<b>11</b>	<b>Keeping Data safe with Encryption</b>	<b>92</b>
11.1	Using PyCryptoDome Library for data encryption . . . . .	92
11.1.1	Hashing . . . . .	93
11.1.2	Encryption with Block Cyphers . . . . .	94
11.1.3	Encryption with Stream Cyphers . . . . .	95
11.1.4	Public Key Encryption . . . . .	95
11.2	Using py-bcrypt Library for Password Hashing . . . . .	98
11.3	Secure login with BCrypt & Flask . . . . .	99
<b>12</b>	<b>An HTTP Server from Scratch</b>	<b>103</b>
<b>III</b>	<b>Appendices</b>	<b>114</b>
<b>A</b>	<b>Cribsheets</b>	<b>115</b>
A.1	Linux . . . . .	115
A.1.1	Some useful aliases . . . . .	115
A.1.2	Some useful commands . . . . .	115
A.2	Vim . . . . .	116
<b>B</b>	<b>Annotated Code Examples</b>	<b>118</b>
B.1	Python3 Flask ‘Hello Napier’ . . . . .	118
<b>C</b>	<b>Additional Miscellaneous (but useful) Tools</b>	<b>120</b>
C.1	cURL . . . . .	120
C.2	Pip . . . . .	120

<b>D</b>	<b>Python Primer</b>	<b>122</b>
D.1	Variables & Data Types . . . . .	122
D.1.1	Variable Assignment . . . . .	122
D.1.2	Performing Calculations With Variables . . . . .	122
D.1.3	Data Types & Type Conversion . . . . .	122
D.2	Help . . . . .	123
D.3	Strings . . . . .	124
D.3.1	String Operations . . . . .	124
D.3.2	String Methods . . . . .	125
D.4	Lists . . . . .	125
D.4.1	List Elements . . . . .	125
D.4.2	List Operations . . . . .	126
D.4.3	List Methods . . . . .	127
D.5	JSON . . . . .	127
<b>E</b>	<b>Git</b>	<b>130</b>
E.1	Bare Repositories . . . . .	130
<b>F</b>	<b>Deployment</b>	<b>134</b>

# List of Figures

3.1	Logging into a remote webtech VM using the command line version of SSH. This will be a similar process across multiple operating systems and is illustrated here using the Windows terminal version. . . . .	12
3.2	The PuTTY Window after you load it . . . . .	13
3.3	The PuTTY Window with completed connection details for the development server . . . . .	13
3.4	The PuTTY alert Window . . . . .	14
3.5	The dev-server login Window . . . . .	14
3.6	The PuTTY logging options. Select “All session output” to keep an automatic log of your session. . . . .	16
3.7	The default Vim Editor window . . . . .	21
3.8	The Vim Editor with ‘:q’ command entered . . . . .	21
4.1	Your first Python Flask web app. . . . .	32
5.1	Flask internal server error . . . . .	37
5.2	An error stack trace example . . . . .	38
5.3	The default 404 Not Found page . . . . .	40
5.4	Custom 404 error page . . . . .	41
5.5	Displaying a static image using the <i>url_for</i> function . . . . .	45
6.1	Result from GET’ing our account/ route . . . . .	48
6.2	Our basic HTML form when GET’ing the account route . . . . .	50
6.3	Page displayed after POST’ing the form . . . . .	51
6.4	Using URL variables . . . . .	52
6.5	Output from using specific URL variable types . . . . .	53
6.6	Output with no URL parameter . . . . .	54
6.7	Output when supplying a ?name=simon URL parameter . . . . .	55
6.8	Inspecting request and response headers using the Chrome Developer Tools . . . . .	59
7.1	Rendered HTML with a very simple template & a single parameter . . . . .	61
7.2	Conditional template rendering without URL arguments . . . . .	63
7.3	Conditional template rendering with a single URL argument . . . . .	64
7.4	Looping over data in within a template to generate HTML . . . . .	65
7.5	The first page that inherits from our base template . . . . .	67
7.6	The second page that inherits from our base template . . . . .	68
7.7	The rendered base template . . . . .	69
9.1	The unstyled HTML page for the Bootstrap example . . . . .	81
9.2	After only including the Bootstrap CSS file . . . . .	82

9.3	Bootstrap example with navigation bar . . . . .	84
9.4	Bootstrap example in responsive mode . . . . .	85
12.1	‘Hello Napier’ from our super simple Python3 HTTP server implementation	106
12.2	‘Hello Napier’ as HTML from our super simple Python3 HTTP server implementation . . . . .	111
12.3	Serving index.html from our super simple Python3 HTTP server imple- mentation . . . . .	113
F.1	The default Caddy welcome web page. . . . .	137

# Part I

## Admin

# Chapter 1

## Introduction

Welcome to the Advanced Web Technologies module from Edinburgh Napier University. This module has a slightly different structure to many modules so it's worth reading through this guidance before you get stuck into the good stuff.

This module is about the Web, what the Web is, what we can currently do with the Web, and what the Web might be like in the future. Rather than focus on the user side of web technology, such as CSS and Javascript, which we've all probably seen in other modules (particularly *web tech*, the pre-requisite for this module), we are going to take a more holistic approach and examine both the server and the client side. We shall look at HTTP (the core of all web technologies) and learn how APIs, Web Services, & RESTful architectures are built to move data around. We will then delve into some more topical discussions, starting with security & privacy on the web, then looking at adding intelligence (Semantic Web), adding increased, scalable interaction (Realtime Web) and private, anonymous, and un-censorable web-technologies (Dark Web). We wrap up the module by examining a few technologies that might form the basis of future web capabilities like Blockchain and IPFS. Throughout the module you should be using the topics as a starting point. A starting point for your own, self-directed, exploration of the topic in more depth. Because we can only really survey any given aspect during contact time; the real knowledge comes from you digging into it in depth. Also a starting point for you to critically reflect on your own experiences and responses to the topics and issues that we raise in class. Whilst the Web itself is a technical mechanism, a tool for moving data around, it also operates in a very complex socio-technical context that currently affects, and will likely affect to an even larger degree in the future, all our lives.

Lectures and labs will not necessarily align neatly, the labs will progress, on a week-by-week and chapter-by-chapter basis to form a first course in “*developing server side web-apps using Python and associated professional practises in a Linux-based environment*”. The lectures will summarise some of this material, providing an opportunity to talk over what we learned in the lab, but primarily, the lectures are an opportunity for us to discuss many other aspects of the Web, focussing in particular on two aspects. Firstly, the structure of the existing Web. Secondly the various facets of Web technologies that influence or alter the way that we use, manipulate, navigate, and share information. These break down into a series of named facets: the semantic web, the realtime web, the dark web, and the permanent web, each of which imposes its own constraints on and proffers opportunities for what the Web is and can do. By looking at each of these in turn we



should gain some insight into how the Web is developing, the directions it might take in the future, and perhaps, suggest areas that we can usefully and positively affect.

# Chapter 2

## Overview

How should we use this workbook? Ideally we would work through it on a chapter-by-chapter basis supplementing our work with background reading and wider exploration of each topic introduced. Some chapters will take longer to complete than others, and other chapters will need to be returned to multiple times. This is particularly true for the first two chapters. To learn both Linux and Python in a fortnight is a tall order so I'd suggest working iteratively, do enough to make some progress, then frequently return to the respective chapters to learn a little more, usually by following the links and footnotes to further practise materials.

In the first week work through the first chapter in the lab section of the workbook. You can read ahead if you want but don't try to run any Flask web-apps on the dev server until you've been assigned your personal virtual server to run your own web-apps on. This week is mostly concerned with the foundation of our learning environment. Logging in, learning to navigate and do simple tasks at the command line, using a non-gui text editor, and using Git. There are links, usually in footnotes, throughout the chapter, for example, to practise the Linux command line then there are online web sites like the *LinuxZoo*<sup>1</sup> that you can use to practise your skills. Similarly, the links to Vim practise tutorials, particularly the Vim game, will help you practise the skills you need to work efficiently in subsequent weeks. Finally, make sure to work through the linked Git tutorials and ensure that you are confident that you understand each of these tools and their place within the learning environment before moving on to subsequent chapters.

Each chapter is meant to cover about an entire week of study, so don't rush through things within the scheduled lab session just to tell yourself the you've done all the work. As I mentioned in the introduction, topics, whether in lectures or labs are meant to be a starting place, a framework to guide your self-directed study, but not the totality of your learning.

In subsequent weeks you will start to build knowledge of the Python language and the Flask library which provides functionality for building server-hosted web-apps. The next chapter, on Python, is meant to cover at least a weeks work and deliberate practise. Mostly that week is concerned with developing basic skills in a new language, Python, which is actually quite a challenge. This is not because Python is particularly difficult but because learning a computer language well takes time and effort and you have to

---

<sup>1</sup><http://www.linuxzoo.net>

start somewhere. Subsequent weeks will require you to work through various chapters of the workbook. You will find that as you progress you will want to skip ahead to different sections, especially once the assessments are released and you want to include specific functionality in your coursework. So after about chapter 3 I expect that many of you will navigate your own path through the remainder of the workbook. The only proviso is that you should aim to have worked through every chapter by the end of the module.

# **Part II**

## **Labs & Practical Work**

# Chapter 3

## Learning Environment Part #1

Because Linux is one of the most widespread operating systems found on web-servers it makes sense for us to use a form of Linux as our *learning environment* for this *Advanced* web technologies module.

There are a lot of different versions of Linux, known as *distributions* or *distros*, and you might have heard or even used some of them, for example, Debian<sup>1</sup> or Ubuntu<sup>2</sup>. Because we could spend an entire year learning about Linux, and still have more to learn, we shall use a deliberately simplified and stripped down installation of a Linux environment, the Ubuntu Server Edition. The main thing you will notice to begin with is the absence of a Graphical User Interface (GUI). Most web servers, in fact most servers in general, are administered from the command-line (or terminal) using text commands so in this module we'll build some experience of this aspect of modern computer use. The reason for this is that the more hardware and software you have installed on a server, then the more there is to go wrong and each place can interact in complex and hard to disentangle ways. If you only have what is absolutely necessary for your server to run and do it's job and nothing more then you have a simpler environment to bug-fix in and one which should be less error prone and more robust.

**VERY IMPORTANT** Do not install a desktop environment or any graphical tools to your learning environment once you have access to it. Our virtual machines are set up to be lightweight and robust. A GUI will use too many resources, impacting upon other users, and will affect the security and backup software that keeps our virtual infrastructure ticking along nicely. If you do do this, then within a few hours I will get an email from Information Services telling me which machine is a problem, and your machine may be wiped and replaced with a fresh copy. So, opt for an easy life and learn to love the command line ;).

We shall also ignore many aspects of working with Linux, such as administration of Linux servers, so that we can focus on using Linux within the context of web-development. Our focus will be on the development of web-apps and an in depth investigation of what happens on the server side. That said, whilst we won't focus as much on the client, there is plenty of scope within the courseworks to put into practise your previous learning from

---

<sup>1</sup><http://debian.org/>

<sup>2</sup>[ubuntu.com](http://ubuntu.com)

the Web Technologies module from last year to make things pretty and provide a better user experience.

There will be a number of core steps involved in getting acquainted with the learning environment:

1. Linux Development Server - Our Ubuntu Linux Virtual Machine which runs on the ENU virtual infrastructure. Each of you will be provided with the address of a specific virtual machine which is your personal learning environment. You will be able to log into this machine using your regular Napier credentials (matriculation number and password) just like you log into regular desktop machines on campus. Your VM is a basic Ubuntu server installation so we will have to install additional software as we go along. However you only have a limited virtual hard-drive so try to restrain yourself from going wild. As a result, you will have access to the *sudo* command to enable you to manipulate your VM as a root user. This means that you can do anything you like to the VM which includes breaking it beyond repair<sup>3</sup>.
2. SSH - To enable us to connect from our Windows desktop machine to our virtual Linux machine
3. CLI - The Command Line Interface (CLI) where we'll type commands
4. Vim - This is our *non-graphical* editor that we shall use to write our source code and to edit configuration files. There are lots of command line editors available. Pico and Nano are alternatives, and you are welcome to use whichever command line editor you like. However, this workbook will assume that your default editor is Vim and so you might have to adjust some commands and procedures as a result of using something different. Note that there is a good reason to have some ability to use Vim; it is one of the most prevalent editors to be found on servers. So if you are dumped on a remote server and need to fix something, Vim is one tool that you can usually expect to find<sup>4</sup>.
5. Git - We shall use Git to store our source code and as part of the “hand in” for our coursework assignments. You will need to put your final project work into a Git repository and zip it up then upload that zip archive to the Moodle link. The reason for this is that there are various levels of internal and external moderation of assignments that occur and having everything in one place makes these moderation processes operate more smoothly. You are also encouraged to make a backup of your repository to another location, e.g. your own laptop, GitHub, GitLab, BitBucket, or wherever. Note that the GitHub and BitBucket interfaces will automatically give you a link to download your repository as a zip file which can be useful at submission time.

---

<sup>3</sup>If you do break your VM then contact me and I will ask Information Services to reset it. This means that everything on your VM that is not backed up will be lost – so keep backups (ideally keep all your work in Git and make copious notes as you go along)

<sup>4</sup>This applies to Unix, Linux, MacOS, etc. servers. Basically nearly all servers, except Windows. If you're on Windows then all bets are off. But who deploys a web site to a Windows server?

By the end of this section of the module you should be able to access your Linux VM by logging in using SSH, navigate around the command-line, use Vim (or similar) to edit your files, and use Git to store your files. This constitutes a core set of tools that should allow you to log into almost any Linux server, whether a very small one running on a Raspberry Pi<sup>5</sup> or a very large one, for example the top 10 fastest supercomputers in the world all run a version of Linux<sup>6</sup>, and feel quite at home very quickly. I am not expecting you to achieve all of this in just a single lab session, but you should accomplish this easily within the lab *and* this week's self-study time.

**IMPORTANT** It is a good habit for you to keep notes whenever you are learning a new tool or environment. I keep a textfile on my computer which also means that I can occasionally copy and paste commands if necessary. I also keep a paper-based notebook for more reflexive thoughts and ideas. This way you won't have to immediately remember how you solved a problem but can look how you did it last time. This makes the whole process much less frustrating. It is how I learned to work with Linux and Git and I still refer to my notes every so often when trying to do something that I rarely do and can't quite remember the syntax for.

## 3.1 Linux Development Server

The university has a heap of server hardware that is used to provide a virtual infrastructure for us. Within this hardware we have a group of virtual machine that has been assembled purely for our use. Our Linux server use the Ubuntu server distribution<sup>7</sup> and has just enough software installed so that we can learn about and get comfortable with a Linux system, but not so many tools that it becomes too daunting. From now on when I refer to the “dev server” I shall mean this Ubuntu Linux server in the university's virtual infrastructure.

Your dev server will give you a launchpad from which to explore the wider world of Unix-style operating systems. If you get comfortable with your Linux server then you shouldn't feel (too) lost trying any other Linux distribution. There are obviously differences between distros, but the the Linux kernel and the GNU userland tools are frequently the core of any distribution.

What you will notice is that your Linux server is command line only. There is no GUI, no mouse, no point-and-click, and no graphical software. Instead we will have to get used to interacting with the computer using text commands. This is because the vast majority of web servers are text-only. Graphical interfaces provide more complexity, more things to go wrong, and require more hardware to support them whereas we want our web-servers to be robust and performant. Consider this an introduction to how the majority of real-world web-servers exist. There is a very good chance that if you work in any software-centric capacity in the future, then you might have to deal with command-line oriented servers, so why not get used to that now? You might like it.

---

<sup>5</sup><https://www.raspberrypi.org>

<sup>6</sup><https://top500.org/lists/top500/2022/06/>

<sup>7</sup><https://www.ubuntu.com>

Because our server is command line only we'll have to learn some new tools, or at least new ways to interact with tools that we might have already seen, but via GUIs. Eventually your development server will include Python, Flask, SQLite, and Vim, and perhaps even some additional tools installed. We will install them as we go along. By default though you have the standard Linux command line tools. When you log in, the text environment that you are presented with is called a shell and the name of the specific shell we use is called BASH. BASH is, in itself, a programming environment which will run 'BASH scripts'. BASH is really useful for whipping up quick bits of code to solve problems, especially those that involve lots of repetition, i.e. the kinds of things that a computer is good at. We will access this server using PuTTY, or any other SSH client using our regular Napier credentials, e.g. your matriculation number and password. You will be able to find out exactly which VM you have access to but all servers will have an address of the following basic form: `webtech-YEARCODE-VMNUMBER.napier.ac.uk`, for example, I will be using `webtech-2223-00.napier.ac.uk` as a demonstration environment that matches your learning environment. Note that you won't be able to access any of the other VMs so there is no point in trying. The list of mappings of VM to matriculation number should be available on the module's Moodle page from week #1. If your matriculation number isn't listed then contact the module leader. The list is generated automatically from enrollment data which isn't always accurate until after the start of the trimester. We'll get into the actual logging in process in just a moment. There are some preliminary things to cover first.

## 3.2 Alternative Learning Environments (optional)

If you have a Mac OS or Linux laptop then you can easily install of the tools that we'll use, e.g. Python, Flask, Vim, etc. locally without needing to use the development server. Talk to the module leader or the demonstrator about this. However you should note that you will need to put your coursework assignments onto the development server for the hand-in and demonstration so you should not ignore it until the deadline because this will be stressful for you.

A second alternative is to install a Linux, like Ubuntu server, into a virtual machine running under VirtualBox on your own laptop. This is quite advanced and will require non-trivial understanding of Linux and virtualisation. For this reason it's not directly supported but is an allowable option for students who are confident of their Linux admin skills. Again though, you will need to put your work onto your Napier VM for demonstration, hand-in, and assessment purposes.

Finally, you can also, if you really must, install Python and the Flask library onto your Windows machine. However, in this case you should note that the Windows command line is very different to that provided in Linux and Mac OS.

In all cases, the alternative learning environments are to help you to find a way to work that suits you, but they aren't a short cut to success.



### 3.3 SSH

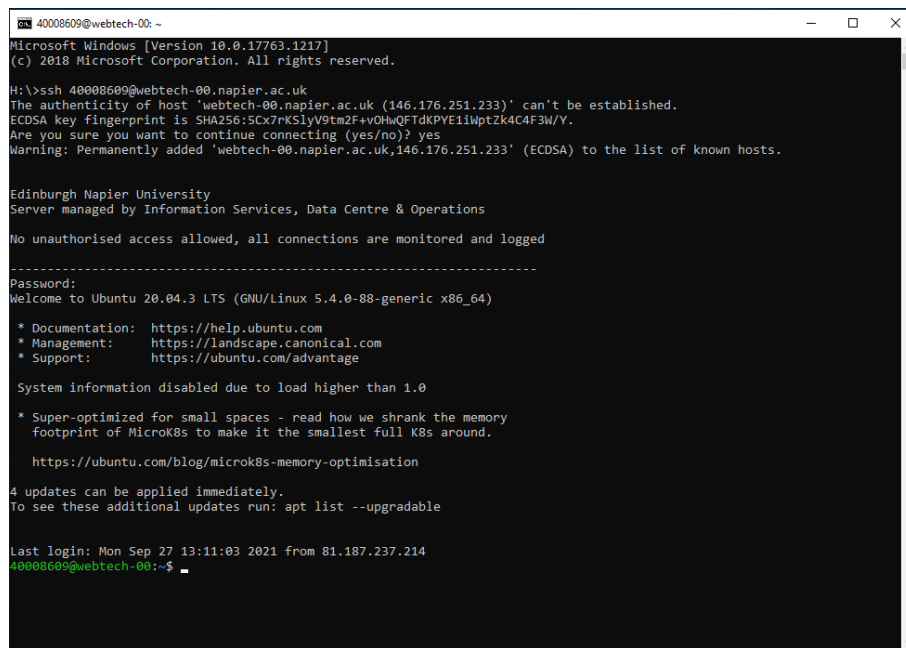
SSH which stands for ‘Secure Shell’ is a tool for logging in to remote servers. This means that you get a window on your local machine into which you can type commands. These commands are executed by the remote server and the results are displayed to you in the window on your local machine. We will use SSH to access the dev server. Note that you can run SSH multiple times to provide you with an easy way to get multiple windows so that you can easily multi-task on the remote machine. For example, one advantage of having multiple shells is that we can edit and save a program in one shell and run the program and monitor output in another shell. As you can imagine, this is very useful when we are developing new web apps on the dev server. There are also tools that will manage multiple SSH connections for you.

If you are on Linux, MacOS, or newer versions of Windows then you will already have a version of SSH installed and all you need to do is open a terminal and type the following to log into your dev server:

```
$ ssh MATRICNUMBER@webtech-YEARCODE-VMNUMBER.napier.ac.uk
```

NB. You *MUST* replace MATRICNUMBER with your actual matriculation number from your napier credentials. You also need to replace VMNUMBER with the specific VM number you were assigned and YEARCODE with the academic year, e.g. 2223 for the academic year starting in 2022 and ending in 2023. Note that we should be very careful to type commands accurately, especially the log in command above, and to also to type our password accurately. If we make too many mistakes then the security service on the server will ban you for a period of time. This is a protection to stop attackers from trying multiple attempts at our passwords in order to gain access to the machine. Currently the security setup is quite sensitive, about 2 failed login attempts can get you banned from logging in for about 5 minutes.

You will be asked for your password but note that the characters of your password won’t be shown on screen when you type it in. In fact nothing is shown. This is expected and desirable behaviour and means that someone who is shoulder surfing you cannot determine even how long your password is merely from viewing your screen.



```
40008609@webtech-00: ~
Microsoft Windows [Version 10.0.17763.1217]
(c) 2018 Microsoft Corporation. All rights reserved.

H:\>ssh 40008609@webtech-00.napier.ac.uk
The authenticity of host 'webtech-00.napier.ac.uk (146.176.251.233)' can't be established.
ECDSA key fingerprint is SHA256:5Cx7rKSlyV9tm2F+vOHwQFTdKPVE1iWptZk4C4F3W/Y.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'webtech-00.napier.ac.uk,146.176.251.233' (ECDSA) to the list of known hosts.

Edinburgh Napier University
Server managed by Information Services, Data Centre & Operations

No unauthorised access allowed, all connections are monitored and logged

-----
Password:
Welcome to Ubuntu 20.04.3 LTS (GNU/Linux 5.4.0-88-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

System information disabled due to load higher than 1.0

 * Super-optimized for small spaces - read how we shrank the memory
   footprint of MicroK8s to make it the smallest full K8s around.
   https://ubuntu.com/blog/microk8s-memory-optimisation

4 updates can be applied immediately.
To see these additional updates run: apt list --upgradable

Last login: Mon Sep 27 13:11:03 2021 from 81.187.237.214
40008609@webtech-00:~$
```

Figure 3.1: Logging into a remote webtech VM using the command line version of SSH. This will be a similar process across multiple operating systems and is illustrated here using the Windows terminal version.

Alternatively, if you're on an older version of Windows or a lab machine that is mis-configured to not enable the SSH tool in the terminal, then you will need to download an SSH client<sup>8</sup>. A popular client for Windows is called PuTTY<sup>9</sup>. There is also a link to a cached copy of PuTTY from the module web page and it should also be available through AppsAnywhere. Download PuTTY, put it somewhere safe where you can access it then double click it to run. There is no need to install PuTTY as it is very portable. When PuTTY runs you will be presented with the following screen:

---

<sup>8</sup>I daresay that with all of the other command line tools and workflows we're looking at in this module, you might also just want to balance things out by using a graphical tool instead. Your choice.

<sup>9</sup><http://the.earth.li/~sgtatham/putty/latest/x86/putty.exe>

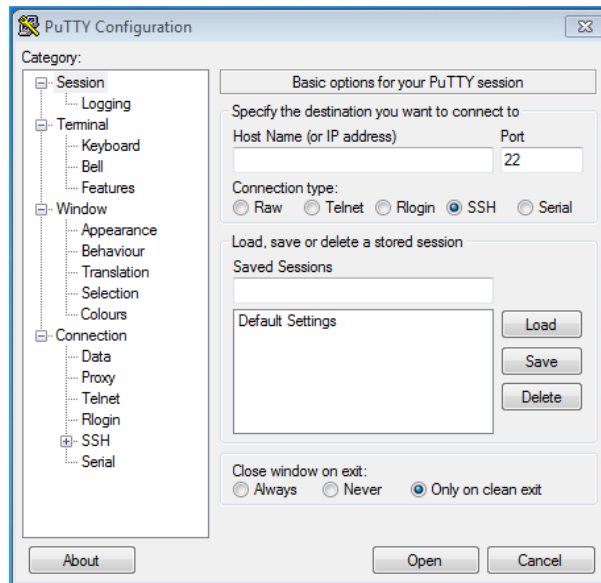


Figure 3.2: The PuTTY Window after you load it

or like this when you've added your VM address to the *Host Name* textbox (again remembering to replace MATRICULATION with your actual matriculation number and NUMBER with your VM ID:

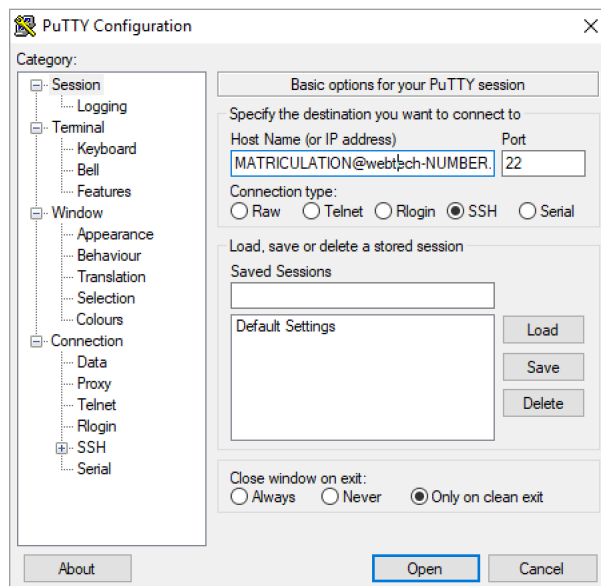


Figure 3.3: The PuTTY Window with completed connection details for the development server

In either case you might be presented with a window similar to the following, especially if this is the first time you are logging into the remote server:

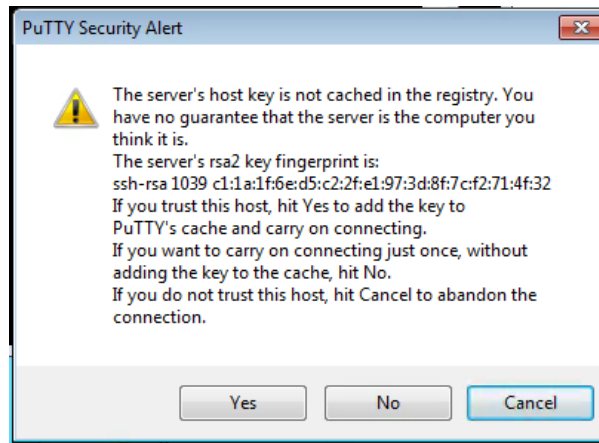


Figure 3.4: The PuTTY alert Window

If so, you can just click the 'Yes' button (but should read and try to understand what it is saying). If all goes well and you are on campus then you will be presented with a window containing a command line interface at which you can work. This Window is a connection directly to your VM and anything you do there is actually happening across the network on a remote piece of hardware. If you are off campus then you will likely be presented with a login window for the dev server and all you have to do is type in your ENU password to authenticate. Again, be accurate so that you don't temporarily lock yourself out. After all this you should then be presented with a command line shell on your dev server. Note that some details may vary depending upon which OS you are accessing your dev server from and depending upon your user name. The following is only indicative of what to expect:

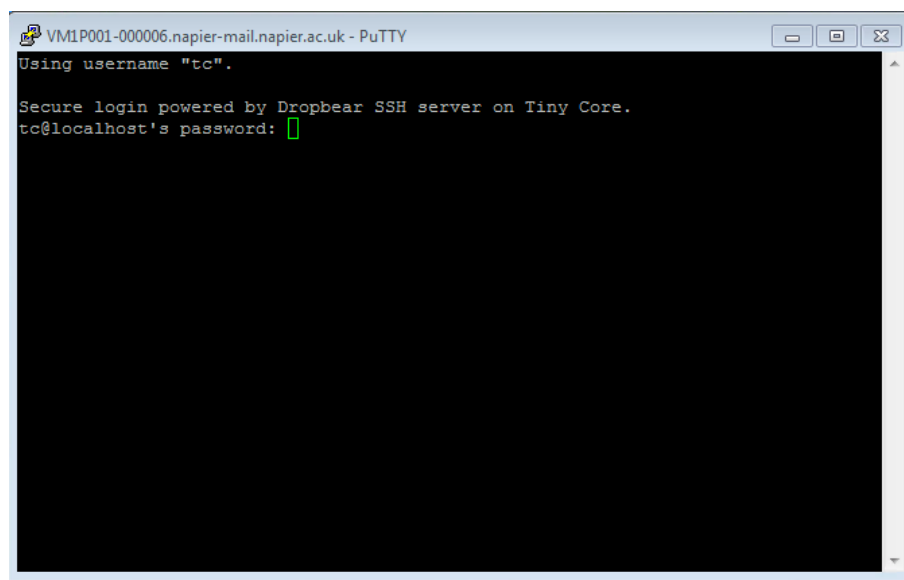


Figure 3.5: The dev-server login Window

Remember, you can repeat this process with PuTTY as many times as required to get yourself enough shells into the development server to make things easier to work with<sup>10</sup>.

<sup>10</sup>This is not the best way to work with multiple shells. Another, more powerful way is to use a shell multiplexer, my favourite is one called Screen (<https://www.gnu.org/software/screen/>) (Tmux is an

Alternatively you can use a different SSH client tool, there are many for Windows and some have useful features, like managing multiple connections. Alternatives include the following, but there are many more, Bitvise SSH Client<sup>11</sup>, MobaXTerm<sup>12</sup>, MremoteNG<sup>13</sup>, secureCRT<sup>14</sup>, and Solar Putty<sup>15</sup>. PuTTY is fairly basic in terms of functionality but others, such as Bitvise and MobaXTerm enable you to create and manage multiple connections to your VM to make it easier to work. For example, having one connection devoted to an editor, another devoted to the CLI, yet another devoted to Git, and perhaps another devoted to executing the Flask debug server. Note that not all of these are free however, but some have a demo, evaluation, or community version that you can use. The moral here is to try things out, see what works for you, and care about the tools that you learn to use.

**IMPORTANT TIP** I usually use SSH to log in at least twice. This gives me a window to edit text in and a window to test my code in. Often I will also have a third window open which I use for command line interaction, Git commits, and testing API calls using cURL, but two is the minimum for a nice development setup - Remember, this is your development environment so you might as well take some time to set it up nicely and so that you can work more productively.

You will, at some point, encounter errors or difficulties whilst logged into your remote machine. In order to get support, and also to provide information to support your helpers, it can be useful to provide information about the commands you have run and the results of doing so. It turns out that PuTTY can help you to do this. PuTTY has a logging option which will automatically create a log file of your SSH session but it is off by default. You can enable logging by clicking the “All session output” option in PuTTY’s configuration. You can find this option in the Category → Session → Logging tab of PuTTY. You’ll then have a file that you can attach to emails when asking for support. Note the other options in the PuTTY interface though, specifically the ones for automatically creating different named log files using, e.g. date, time, and hostname information, so that your session log is automatically named with a sensible identifier.

---

alternative that is popular) but these are yet more tools to learn, and another set of commands, so let’s try to keep things *reasonably* simple for now. Being aware that these things exist though means that when you really need them you know that there is a tool out there waiting for you.

<sup>11</sup><https://www.bitvise.com/index>

<sup>12</sup><https://mobaxterm.mobatek.net/>

<sup>13</sup><https://mremoteng.org/>

<sup>14</sup><https://www.vandyke.com/products/securecrt/>

<sup>15</sup><https://www.solarwinds.com/free-tools/solar-putty>

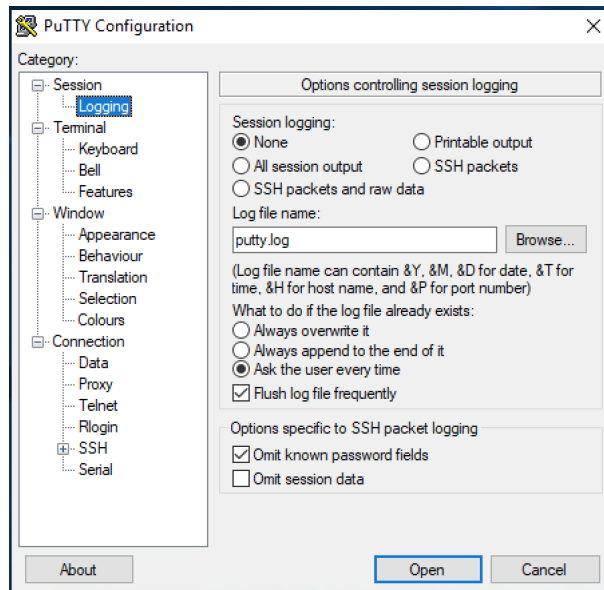


Figure 3.6: The PuTTY logging options. Select “All session output” to keep an automatic log of your session.

## 3.4 Basic Linux Usage

When we first log in to Linux we will see a prompt, a place where we can type commands. It looks something like this (again, your specific VM might have a different name so might not *exactly* match this):

```
40001111@soc-web-liv-12:~$
```

This simply means that the user ‘40001111’ is logged into the machine called ‘soc-web-liv-12’. The ‘.’ is merely a separate between the user@machine part of the prompt<sup>16</sup> and the next part ‘~’ which is your current location in the filesystem. The tilde or ‘~’ symbol is used on Linux machines to mean your home directory. Finally the ‘\$’ symbol is your “prompt”, the place where anything you type will be displayed.

Note that in the remainder of this workbook, we’ll just show the ‘\$’ part of the prompt to simplify the examples.

Linux has a file system, a way for all of the files that make up the running system to be organised hierarchically, just like Windows. However, on Linux the file system is organised a little differently. Instead of starting at ‘C:’ the Linux file systems starts with ‘/’ which is also known as the “root” of the filesystem. The word root is used because the Linux filesystem is shaped like a tree. All of the resources that you can access, such as your own files, are located at some level somewhere within the tree. An advantage of the Linux approach is that, when you add extra hard-drives, or mount network resources, instead of extra drive letters, all of your resources get mounted within the tree, e.g. /volumes/web might be the path to a remote web server and /media/cdrom might be the path to

---

<sup>16</sup>You can customise your prompt to display just the information that you want to be displayed, but this is left as an exercise for you

your CD-Rom drive<sup>17</sup>. But other than that, from a basic user-oriented perspective, both Windows and Linux file systems are a hierarchical collection of files and directories in which any given directory might contain zero or more files or child directories.

When you log in you will be located in your own directory, called your “home” directory which is located in the filesystem tree at

```
/home/YOUR-MATRIC-NUMBER
```

You can always see where you are in the filesystem by typing

```
$ pwd
```

which is short for “print working directory” or “tell me where I am”. The `pwd` command will give you the full hierarchical list of folders, from the root, to your current location in the file-system. You can type this anywhere you have a prompt. Also, no matter where you navigate to within the file system you can always return to your home directory by typing:

```
$ cd
```

You can navigate around the file system using the `cd` command which is short for ‘change directory’<sup>18</sup>. The default version, without an argument takes you home, as we said before, but if you supply an argument then you can change the current directory. Let’s try that now by changing to the root of the filesystem:

```
$ cd /
```

If you now use `pwd` you should see that the output is different to what it was before. You are no longer in your home directory but are in the root directory instead. Now use the `cd` command without an argument to go home and use `pwd` to see where you are again. You can also step up through the directory hierarchy by using the ‘..’ argument:

```
$ cd ..
```

‘..’ is an *alias*, a label that has a default meaning, which in this case means “move into the parent directory”. Explore the filesystem for a while using the `cd`, `cd /`, `cd ..`, and `pwd` commands.

---

<sup>17</sup>if you still have such an old-fangled device

<sup>18</sup>You’ll notice that many Linux commands are shortened versions of longer words. This is partly designed to reduce the amount of typing that you do. It may seem silly now but when you are changing directory hundreds of times a day it is much nicer to type `cd` than `change-directory` each time.

There is a limit to what we can do with just these commands because we can only move into our home directory, or else navigate up through the tree to the file system root directory. We need a couple more commands to let us see what is inside a directory and to move into a new directory. For this we use the 'ls' command which is short for list or list contents to see what files or child directories are in our current directory, and the cd command but with the name of a child directory as the argument. So, we can use 'ls' as follows:

```
$ ls
```

to list the files in the current directory. If you try this now you will be able to the files in your own home directory. There will probably be none at this point but don't worry, that will soon change. My test account for the dev server gave the following output:

```
$ ls
test  test.py
```

Now we can create new files in our home directory quite easily using touch, e.g.

```
$ touch testfile.txt
```

which should give us something like this:

```
$ ls
testfile.txt
```

We can also create new directories using the 'mkdir' which means make directory

```
$ mkdir testdirectory
```

which results in

```
$ ls
testdirectory/  testfile.txt
```

We can also move files around using the 'cp' and 'mv' commands which are short for "copy" and "move" respectively. Let's see them in action; first we will make a copy of testfile.txt then move the copy into the testdirectory:

```
$ cp testfile.txt testfile2.txt
$ ls
testdirectory/ testfile2.txt  testfile.txt
$ mv testfile2.txt testdirectory/
$ ls
testdirectory/  testfile.txt
$ ls testdirectory/
testfile2.txt
```



Notice that in this example we passed an argument, the name of a directory ‘testdirectory’ to the ls command and this caused the contents of testdirectory to be listed instead of the current directory. We can do that with any directory that we have access to.

Finally we might want to delete files to keep things tidy. We can use the ‘rm’ command to remove files, e.g.

```
$ rm testfile.txt
$ ls
testdirectory/
```

We can use also use rm to remove directories, however, by default we get this behaviour:

```
$ rm testdirectory/
rm: cannot remove 'testdirectory/': Is a director
```

What we need to do instead is to supply some options to the rm command. We need rm to act recursively, that means move into the specified directory and delete its contents and we also need to force rm not to stop and ask us for each file whether it should be deleted. We there fore need to use rm as follows:

```
$ ls
testdirectory/
$ rm -rf testdirectory/
$ ls
```

Of all the commands we have met so far, rm is the only one that can do any real damage. rm -rf could conceivably delete all of your files and directories if the command is executed in the wrong place. As a result you should, especially once you start writing code on the dev server for your assignments and lab exercises, keep backups of anything that you will need to use again. This goes twice for the code for your coursework assignments. Note that one of the tools we’ll learn later, Git, is specifically designed to help you back up your code to other locations, such as GitHub or BitBucket so the best thing to do is to learn to use Git well rather than looking for non-standard alternatives.

There are many, many more commands than just these. In fact you can do much more with the command line than you can with graphical tools. However, this should be enough for you to get started and is enough for you to be able to create and delete files and directories, to navigate the file system hierarchy, to list the contents of directories, and to view the contents of files.

To do more exploration of Linux, you can of course experiment on the dev server. Your account is quite limited in what it can do so you shouldn’t be able to do too much damage. Another good place to start is the Linux Zoo site<sup>19</sup> which offer online virtual machines, more information, and a number of tutorials. Particularly the Linux Zoo “essential Linux”

---

<sup>19</sup><http://www.linuxzoo.net>

pages<sup>20</sup> which explain in more depth some of the tools that we have already covered plus many many more.

## 3.5 Vim

Vim is a command-line based text editor. It is based on an earlier editor called Vi (Vim is VI improved, hence Vim, which is a little easier to use). You might ask why we don't just use a GUI text editor like Notepad, or the editor in Visual Studio. The main reason is because we are talking about advanced web technologies, and dealing with them often involves accessing remote servers. Furthermore, the majority of servers do not have a graphical interface and only have installed the most minimal and robust set of tools. The one editor that you are almost guaranteed to find on any Unix or Linux server is Vi and by learning Vim you are well placed to handle Vi. As a result it makes sense for us to become familiar with it. Many of us actually find that the shortcuts and powerful control that Vim offers us mean that we just concentrate on learning one editor very well and only use that editor rather than moving to a different editor each time we need to write a different type of document or program in a different language. That all said, if you really struggle with VIM, then your VM already has the slightly simpler command line editors, nano and pico installed. You can run either by typing it's name at the terminal. Note that these have an on-screen menu to remind you of the core editing functions available and they are accessed via a key combination such as ^X, which is the combo for exiting nano (The '^' in this case means the ctrl key on your keyboard and not the caret sigil above the number 6).

What makes Vim different from many of the editors that you will be familiar with, like Notepad, is that there is no role for the mouse, no buttons to click at all, only keyboard shortcuts, so we shall learn just enough of those in this module to be able to edit basic documents<sup>21</sup>. A second very important aspect of Vim is that it is a *modal* editor. When you use Vim you use it in different modes, when you are typing content into a file then you are in edit mode, and when you are entering commands you are in command mode. When Vim starts it is in command mode and anything you type will be interpreted as a command for Vim to perform. You switch into edit mode by typing 'i' (for insert) and you can return to command mode at any time merely by hitting the escape key.

**IMPORTANT** If you are ever unsure what mode you are in the you can just hit escape a couple of times to ensure you are in edit mode. From here you can just type 'i' again to enter the edit mode.

Start Vim by typing Vim at the prompt:

```
$ vim
```

---

<sup>20</sup><http://linuxzoo.net/page/intro.html>

<sup>21</sup>However there are hundreds of Vim commands and a really clever thing is that most of the commands can be chained together so that you can automate many editing tasks.

and you will see something similar to the following:

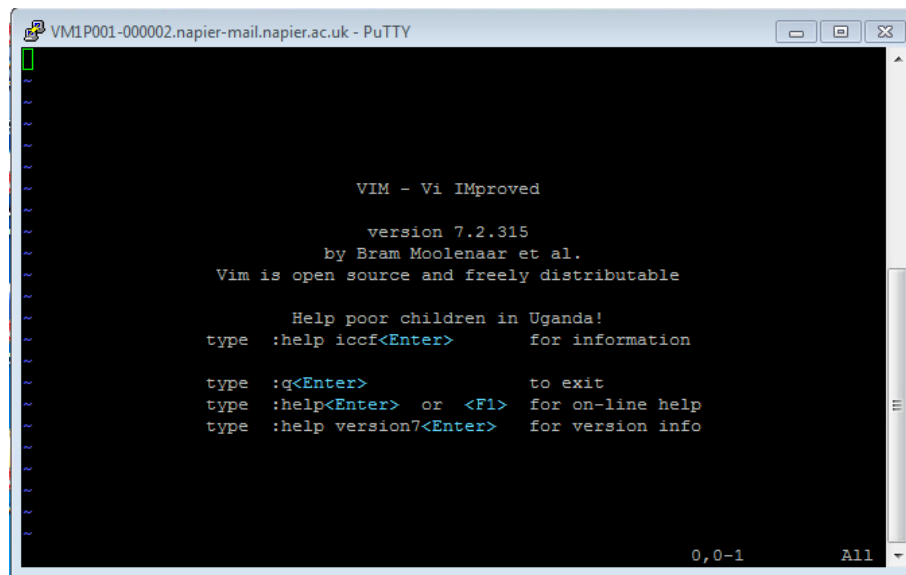


Figure 3.7: The default Vim Editor window

Let's do the easiest thing first. Let's quit Vim. To do this we enter the command mode by hitting escape then enter `:q` (where `q` is short for quit) and hit enter, e.g.

```
<ESC>:q<ENTER>
```

Your Vim window should look something like this when the command is entered (but before you press enter):

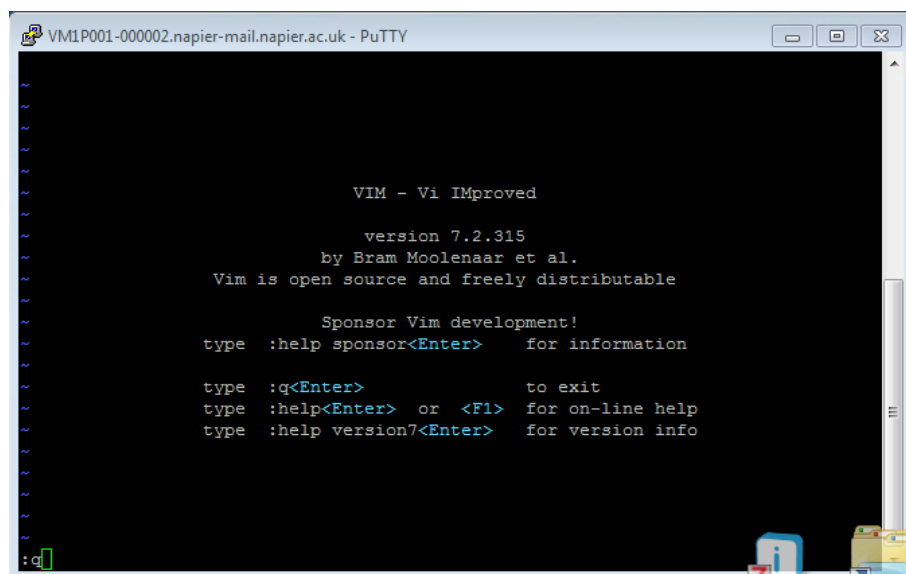


Figure 3.8: The Vim Editor with `:q` command entered

Once you hit enter you will be dumped back to the Linux prompt. Two other useful commands to be aware of for quitting are as follows:

- To quit and discard any changes, i.e. if you have already made changes to a document that you don't want to save:

```
<ESC>:q!<ENTER>
```

- To quit and save changes, we use :wq for (w)rite and (q)uit:

```
<ESC>:wq<ENTER>
```

Let's start Vim again and actually edit some text. This time when Vim starts you need to press 'i' for (i)nsert to enter the edit mode. You can now type away to your hearts content. When you are ready to save the file you can enter the command mode and type :w for (w)rite. If the file doesn't have a filename you will get a message to that effect so, with a new file that isn't yet saved you can use :w filename.txt (where filename.txt is the name of the file that you want to create). This file will then be created in whichever directory you were in when you started Vim.

There is **A LOT** to learn in Vim. All those shortcuts are really a language for manipulating text. The easiest way to learn Vim, just like any new tool, is to use it. There are also many online resources that teach you how to use Vim but two of my favourites are:

1. The Open Vim Tutorial: <http://www.openvim.com/>
2. Vim Adventures: <http://vim-adventures.com/>

I recommend visiting those site and particularly trying some of the exercises to get used to some of the core Vim functionality. There is also a cribsheet of useful commands in section A.2.

## 3.6 Git

Git is a source control system that enables you to keep track of your source code, its history and any changes you make. Git can be used to track any file but is most efficient and best suited when used only with textual files. Because Git is a *distributed* source control system it works very well to enable groups of people to work on the same source code as well as supporting experimenting with your code, trying out lots of different ideas in separate *branches* (which are a bit like a copy of your code but with tools to help manage that copy and support re-integrating it with your main source tree if you want to), and being able to roll back to an earlier version if you decide you have taken a wrong turn.

Whilst you might have seen Git before, perhaps as a plugin to an IDE or as a standalone GUI app, Git is primarily a command line application, so we shall use it to get our source code in and out of the dev server.

**IMPORTANT** We shall use Git to support the hand-in of courseworks in this module so you should get familiar with it as soon as possible. A good place to start is by dipping into the Git SCM book<sup>22</sup>.

---

<sup>22</sup><http://git-scm.com/book/en/v2/Getting-Started-About-Version-Control>

There are also numerous interactive tutorials and resources to help you get started with Git:

1. The Git Parable: <http://tom.preston-werner.com/2009/05/19/the-git-parable.html>
2. Github's Learn Git 15 minute tutorial: <https://try.github.io/levels/1/challenges/1>
3. Learn Git Branching <http://pcottle.github.io/learnGitBranching/>
4. Git Immersion [http://gitimmersion.com/lab\\_01.html](http://gitimmersion.com/lab_01.html)

You should also create either a Github account<sup>23</sup> or a Bitbucket account<sup>24</sup> (or both if you like) then create a repository within your new account called 'set09103'. You will push all of your code throughout the module into this repository and at hand in time I will pull a copy for marking. The advantage of this approach is that at any point, if you need help with your code, then we have a copy that I can see remotely. However this only works if you keep adding your code to your repository. That means whenever you make changes you need to (1) add them, (2) commit them with an explanatory message, and (3) push the changes from your local repository to the shared one on Github or Bitbucket.

Git is already installed in our learning environment. In order to use it we have to do a couple of things. Log into the dev server then do the following (obviously replace my email address and name with your own):

```
$ git config --global user.email "s.wells@napier.ac.uk"
$ git config --global user.name "Simon Wells"
```

It is easiest to create a new repository through the interface on Github or Btbucket then use the repository cloning address to make a local copy as this sets up all the addresses automatically for pushing and pulling code. Once a repository is set up and you have cloning address you can do something similar to the following (where <https://siwells.github.com/siwells/sandpit.git> is the name of one of my repositories on github, your's will obviously be different):

```
$ git clone https://siwells.github.com/siwells/sandpit.git
```

We can then make changes within the new local clone of, in this case the 'sandpit' repository, then add, commit, and push as follows (again the details of adding and committing depend upon the exact files that you have altered. In this case we'll assume that a file test.txt has been edited):

```
$ git add .
$ git commit -m "Added introductory example"
$ git push
```

---

<sup>23</sup><https://github.com/>

<sup>24</sup><https://bitbucket.org>

Again, as for Linux and Vim, there are many options and powerful features that you can learn to use with Git, however we shall try to keep things as simple as possible for now.

**ADVANCED** If you are comfortable with using Git and SSH then you might want to try setting up SSH keys so that you don't have to use a password each time you push code into your remote repository.

## 3.7 Wrapping Up

Obviously this has only been the most basic of introductions to web development using a Linux server. There is much much more that you could learn about any one of the tools that we have introduced and it is well worth your time to explore additional resources and reading for each of them. It is very likely that you will experience some or even all of these technologies, in some form, at some point of your career and putting in some extra effort now will mean that you are much more capable later.

# Chapter 4

## Learning Environment Part #2

After last weeks extravaganza of new tools, we will only introduce two new tools this week, the Python language and a Python Library for developing web applications called Flask. By the end of this weeks practical work you should be able to build a simple ‘Hello Napier’ web app using Python and Python-Flask which is hosted on and runs within our learning environment and whose source code is pushed to our personal Github (or Bitbucket) repositories.

In total, by the end of this week we should all have at least some rudimentary experience with each of the following software tools:

1. Linux Development Server - Our Linux server which runs on the ENU virtual infrastructure.
2. SSH - To enable us to connect from our Windows machine to our virtual Linux machine
3. CLI - The Command Line Interface (CLI) where we’ll type commands
4. Vim - This is our *non-graphical* editor that we shall use to write our source code.
5. Git - We shall use Git to store our source code and to “hand in” our coursework assignments
6. Python - A general purpose programming language that is well suited to web development.
7. Python Flask - A Python library for server-side web-app development.

This constitutes our entire learning environment. If you have missed any part of the learning environment then it is worth going back and filling in any gaps as they all work together to give us a solid foundation for the development of professional development skills and the production of robust and scalable web-apps. In other words....

**VERY IMPORTANT** The work this week builds directly on last week so if you haven’t finished working through chapter 3 then it is best to do that first or you will probably get horribly stuck.

## 4.1 Python

Python<sup>1</sup> is a very useful programming language and much of its popularity stems from the fact that it is easy to get a lot done with having to write too much code. We already have Python installed on the dev server ready to use. We can run Python by typing its name in the terminal, e.g.

```
$ python3
Python 3.8.2 (default, Jul 16 2020, 14:00:26)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

This starts the Python Read-Evaluate-Print-Loop or REPL in which we can type Python commands and get immediate output. To exit the REPL we type ‘quit()’ which will return us to the Linux shell, e.g.

```
$ python3
Python 3.8.2 (default, Jul 16 2020, 14:00:26)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>> quit()
$
```

Here is the traditional ‘Hello Napier’ program in Python (try it out for yourself):

```
$ python3
Python 3.8.2 (default, Jul 16 2020, 14:00:26)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>> print("Hello Napier")
Hello Napier
>>> quit()
$
```

The REPL is very useful for trying out bits of Python as you are learning and also for interactively developing code and analysing data. There are super-charged REPLs and related tools such as iPython<sup>2</sup> and iPython notebooks<sup>3</sup> which are used in data science and data analysis. However, once we are comfortable with the Python syntax, we will mostly want to run Python scripts, so let’s do that.

Create a new file called first.py, open it in Vim then edit it as follows:

```
1 print("Hello Napier")
```

---

<sup>1</sup><https://www.python.org/>

<sup>2</sup><http://ipython.org/>

<sup>3</sup><http://ipython.org/notebook.html>



Save and close first.py then execute the following command in the shell:

```
$ python3 first.py
Hello Napier
```

Congratulations. Another first. You just ran your very first Python script. All we will really do from now on in the remainder of this module is build on top of this basic script in order to build our own web-apps. However, we shall have some help along the way from some really excellent libraries so don't worry, we shan't build everything from scratch.

Now the best thing to do is to learn some Python syntax by exploring some of the excellent online 'learn Python' tutorials that are available. Some of them are interactive, so you can type directly into the web-site and get results. Others just give you exercises to do yourself and you can do those exercises in Python on our dev server platform. Here are a few, in order of usefulness according to me, but many more are just a Google search away:

1. Learn Python the hard way: <http://learnpythonthehardway.org/book/>
2. The Python Practice Book: <http://anandology.com/python-practice-book/index.html>
3. A Byte of Python: <http://www.swaroopch.com/notes/python/>
4. Code Combat: <https://codecombat.com/>
5. Python for you and me: <http://pymbook.readthedocs.org/en/latest/>
6. The Hitchhiker's Guide to Python: <http://docs.python-guide.org/en/latest/#the-hitchhiker-s-guide-to-python>
7. Hands-On Python Tutorial: <http://anh.cs.luc.edu/python/hands-on/3.1/handsonHtml/index.html>
8. The Python Challenge: <http://www.pythonchallenge.com/> - Some quite touch challenges that you can solve using Python (NB. Assumes you already know what you are doing)
9. Python Tutor: <http://www.pythontutor.com/> - Helps visualise the execution of Python code. Again, assumes that you have some prior knowledge of Python syntax.

The links towards the top of the list are aimed at those completely new to the Python language whereas those further down the list will help those who have some Python knowledge already. Another good way to practice a new language or to improve your existing abilities, not just in Python, but in any language, is to try to solve a set of problems using the language. I often use Project Euler when starting out with a new language but there are also many others:

1. Project Euler: <https://projecteuler.net/>
2. Stack Exchange Code Golf: <http://codegolf.stackexchange.com/>

3. Code kata: <http://codekata.com/>
4. Reddit Daily Programmer: <https://www.reddit.com/r/dailyprogrammer>
5. Programming Praxis: <http://programmingpraxis.com/>
6. Rosetta Code: [http://rosettacode.org/wiki/Main\\_Page](http://rosettacode.org/wiki/Main_Page)
7. International Collegiate Programming Contest Problems Index: <http://acm.hit.edu.cn/judge/ProblemIndex.php>
8. Algorithmist: [http://www.algorithmist.com/index.php/Main\\_Page](http://www.algorithmist.com/index.php/Main_Page)

It is also worth creating a git repository of any code that you create when learning a new language because if you take a break from that language to do something else, you will have somethings to remind you of where you were up to before. It is also a neat way to share your solutions with others. If you exhaust all that lot then another great practice approach is to try to implement your own versions of basic (and advanced) algorithms and data structures. For example, if you really want to challenge yourself, and your knowledge of a programming language, read about and implement probabilistic datastructures like Bloom Filters which I find quite interesting, or Fountain Codes which are used in tools like Bittorrent.

## 4.2 Python-Flask

Python-Flask, or just Flask<sup>4</sup>, is a Python based microframework for developing websites in Python. Flask is already installed and ready to run. We shall leave all of the nitty gritty of installing and administrating software on a Linux server as an exercise that is outside of the remit of this module and just focus on using the tools that we already have installed.

Flask includes a lot of useful tools; a development server and debugging tools so that we can run our flask apps during development without needing to run a separate external server like Apache<sup>5</sup> or NGinX<sup>6</sup>. Obviously if we were deploying our apps in the real world then we would host our web-apps differently. We would use a secure Linux install on a server with access to the external world and would use a good HTTP server, like NGinX, and a WSGI-compliant app-server, like uWSGI<sup>7</sup>. We might even use a load balancer like Haproxy<sup>8</sup> to enable us to scale our operations. We will look at all of these aspects later in the module but for now we shall concentrate on the development server and a simplified development environment. Flask also has integrated unit-testing support, RESTful request dispatching, templating, to help generate HTML pages, using the Jinja2 tool.

---

<sup>4</sup><https://palletsprojects.com/p/flask/> - This is the main project website for Flask and should be your first port of call for additional information and documentation

<sup>5</sup><http://www.apache.org/>

<sup>6</sup><https://www.nginx.com/resources/wiki/>

<sup>7</sup><https://uwsgi-docs.readthedocs.org/en/latest/>

<sup>8</sup><http://www.haproxy.org/>

Flask supports us in building web-apps that conform to the Python Web Server Gateway Interface (WSGI)<sup>9</sup>. The goal of WSGI is to make it easy to develop and deploy Python web-app, regardless of the library that is used. So developers can find a library that they find useful, or helps them to tackle their development task, but the output of the process, the web-app has known features and capabilities which means that any server that supports the WSGI can run the web-app. This is similar to what happened with web-app development in Java, where the servlet API means that many tools can be used to build a Java web-app and many web-servers can then run the app.

### 4.2.1 Installation

We should only need to go through this process once. We are going to install some bits of software to enable us to manage python libraries. Flask is a library, but not the only one that we'll use, so having a way to manage different versions of our libraries is a useful thing. It's also the 'proper way to do things'. We will use a program called 'Pip' to install and manage libraries of Python sourcecode, such as the web library (Flask) that we're using in this module. Pip (or rather 'pip3' is already installed on our NapierVM alongside Python so we don't need to install that. However, We will also use a second Python tool that is used to manage sets of Python libraries, so that we can use different versions of libraries for different projects without them all interfering with each other. This tool creates a 'virtual environment' for our python web apps to run in and makes sure that the right versions of the right libraries are available to our web-app. The tool we'll use is called 'VirtualEnv' and we can install it like so:

```
$ sudo apt install virtualenv
```

Again you might be asked for a password and whether you want to continue. Enter your password and type 'y' when, or if, prompted. When this has completed successfully, we should have our prompt back and both the pip and virtualenv tools will now be available. Each can be invoked by typing its name at the prompt. They won't do much yet though.

We can create a virtualenv using the virtualenv command and passing it the name of the environment we want to create. For example, to create an environment called 'test' we could do the following:

```
$ virtualenv test
```

Once a virtualenv is created we need to activate it, like so:

```
$ source test/bin/activate
```

This basically changes where Python looks for the libraries that it can use. We can also use the pip freeze tool to see what libraries are installed, like so:

```
$ pip3 freeze
```

---

<sup>9</sup>The WSGI standard is described in this Python Pep:<https://www.python.org/dev/peps/pep-0333/>. A PEP is a Python Enhancement Proposal and is the main way that the planned development of the Python language progresses.

I don't see Flask amongst the list of installed libraries. Perhaps we'd best install it? We can do that using pip, like so:

```
$ pip3 install Flask
```

If you run pip freeze again you will notice that a bunch of new libraries are installed, these include both Flask and a bunch of supplemental libraries that Flask uses.

When we are done working within a Python virtualenv we can close it down by using the deactivate command like so:

```
$ deactivate
```

You will need to activate and deactivate your virtualenv whenever you want to run your web-app (or use any of the libraries installed within the virtualenv). When we come to deploy our web-apps for live consumption on the web, we'll set things up to work automatically though, but that comes later. Note that you should only need to install pip and virtualenv once. They will then be installed and available permanently on your VM. You can then create as many virtualenv's as you want or need. One for each project or experiment seems like a good rule of thumb.

To manage virtualenvs I usually create a folder for each of my Python projects. I then navigate into that folder and create a new virtualenv within it. I usually name each virtualenv as just 'env'. When I want to work on a project I just cd into the project folder then 'source bin activate' to start the environment. From this point on, until I type deactivate, Python within the current window will use the activated virtualenv. Note that you can run multiple virtualenvs but they must be in different login windows<sup>10</sup>. This will all seem complicated for now, but you will get used to it and it gives you much more control over your development projects as things develop. You can, and should, read more about virtualenvs. One place to start is in the "Hitchhikers Guide to Python"<sup>11</sup>.

Assuming that your virtualenv is activated, and that you've got Flask installed (re-activate your virtualenv if not), you can check whether a library, like Flask, is installed by running the following:

```
$ python3 -c "import flask"
```

This just invokes Python and tell it to use the 'compile' mode and to take the code that is passed in as an argument, in this case just the Python code to import the flask library. Importing is just our way to tell Python that we want to use a particular library for our code. If the library is available then there will be no output, so if there are no errors or other messages displayed then everything is fine and you can move on. If you do

---

<sup>10</sup>or screen or tmux buffers

<sup>11</sup><https://docs.python-guide.org/dev/virtualenvs/#lower-level-virtualenv> - This whole section should be quite interesting and is well worth reading, especially the bit about using pip freeze to create a requirements.txt file, basically a list of libraries that your code uses, so that you anyone who you give your code to can recreate the environment to run it in. Really useful, and efficient because we then don't need to distribute the libraries that our code needs to run, just the code itself.

get errors or messages then it might be worth doing some background research to increase your knowledge before asking for help.

## 4.3 Python Flask “Hello Napier”

Now we can get on with actually building our first web-app using Python and Flask.

Here is our first Python Flask app. You’ll find an annotated version in Appendix B.1 that explains what each line is doing, but see if you can work that out yourself first:

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

**IMPORTANT** Don’t just copy and paste code from the PDF. This is for two very good reasons. The first is that Python is ‘white-space sensitive’. This means that Python uses whitespace as part of the layout of code, instead of using things like curly braces such as ‘{’ and ‘}’. So if you get spaces, tabs, and other whitespace characters mixed up in your file, which can easily happen if you copy and paste, then it will affect the indentation of the file (although perhaps not in a way that you can tell visibly because whitespace can all look pretty much the same). The second very good reason is that typing in the code counts as ‘deliberate practise’. You are more likely to remember stuff if you do it multiple times and typing it in the first time counts as the first opportunity to practise.

Type in the ‘Hello Napier’ code from above or else get it from the module’s Git repository. Save the code in a file called *hello.py*

We can now run our code by executing the following command in the same directory where *hello.py* is stored:

```
$ flask --app hello run --host=0.0.0.0
```

The host argument (‘--host=0.0.0.0’) merely causes our web-app to be available publicly. If we didn’t include this then we would only be able to access the web-app from within the development server. Overall, this command causes our web-app to be run using the Flask development server and to be made available to be accessed from any IP address on the Internet. The Flask development server is really useful and fast for debugging during development. If everything goes well then you should see output similar to the following in your terminal:

```
$ flask --app hello run --host=0.0.0.0
* 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
```

Congratulations. You now have your first Python3 web-app running. Flask will now continue to run in your terminal until either an error occurs or you stop it. To stop it you can send the “SIGINT” or ‘interrupt Signal’ to flask to tell it to stop itself by using CTRL+c<sup>12</sup>.

When we run a Flask web-app, that app will be accessible in a web browser on a particular network port. By default, Flask runs on port 5000. Your web browser only looks to port 80 by default so we need to tell our browser where to find our app on the given server. You can visit the page generated by Flask in a web browser. Just start up a browser in your host OS, e.g. in Windows when accessing the dev server from a lab machine. Then visit the following address `http://webtech-NUMBER.napier.ac.uk:5000` where you have replaced ‘NUMBER’ with your assigned VMs number.

If that address didn’t work then you should also try the IP address given in the output from Flask, e.g. ‘`http://146.176.251.30:5000`’. This is because sometimes the university systems don’t properly have the DNS systems set up correctly to do the lookup from the URL, ‘`http://webtech-NUMBER.napier.ac.uk:5000`’ to the IP address. So we can cut out the middleman and head straight to the IP address instead.

Either way, all going well, you should see something similar to the ‘hello world’ app shown in Figure 4.1.

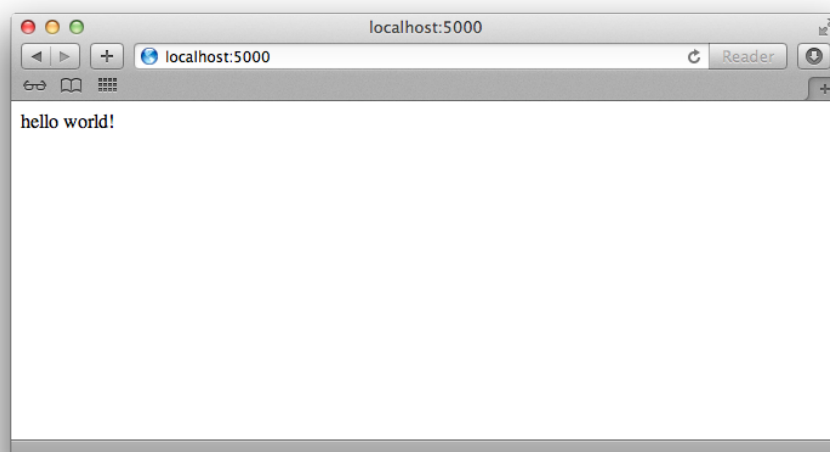


Figure 4.1: Your first Python Flask web app.

---

<sup>12</sup>This means holding down the CTRL button then pressing the C button whilst still depressing CTRL then releasing both buttons.

Well done! You're now a dynamic web-app developer<sup>13</sup>. The remainder of the workbook will basically iterate on this simple web-app, exploring different facets of HTTP and implementing those facets using Python and Flask.

## 4.4 Reading & Resources

This chapter only scratches the surface of Flask. You will find much more information at the Flask project's web page. There are also several sites that explain aspects of Flask and might prove useful. Even if you end up reading the same information a few times, you might find that some voice speak to you better than others:

- The Flask Documentation<sup>14</sup> - The first place to go to for information about the Flask microframework.
- Full Stack Python Flask Page<sup>15</sup> - A great list of online resources for Python Flask
- How a Flask app works<sup>16</sup>
- Miguel Grinberg's Series of articles on Python Flask<sup>17</sup> - This became a published book a few years ago and is well worth getting hold of for an end-to-end demonstration of the development of a whole Flask web-app.

## 4.5 Wrapping Up

Right. That's all the preliminaries in place for building web-apps in our learning environment. We can now run a small and lightweight, but otherwise similar to full Linux installs, virtualised server which will host our web-apps. We can log in to the server using SSH, navigate the Linux environment, and edit files using VIM. We can also take advantage of the installed Python language and Python-Flask web-app library to build our own web-apps.

**CHALLENGE #1** You should already know how to do this, and this is, as a result, merely some extra practise. Create a Github or Bitbucket account. Bitbucket allows you to create an unlimited number of private repositories if you use your @napier address to register whereas Github restricts the number of private repos. That said, either site allows you to create as many public repositories as you like. Create a remote repo in your Github or Bitbucket account called 'set09103'. Log into the Linux dev server then clone your set09103 repo. Create a new file in your repo called 'hello.txt' and put the message "Hello Simon" in it then add, commit and push your changes to your remote repo. Once you have done this, check on either Github or Bitbucket to ensure that your text file is in the repo where you expect it to be then send an email to Simon containing the name of your account, which service, Github or Bitbucket, you are using, and the SSH clone url for your set09103 repository.

---

<sup>13</sup>albeit of a very simple dynamic web app

<sup>14</sup><https://palletsprojects.com/p/flask/>

<sup>15</sup><https://www.fullstackpython.com/flask.html>

<sup>16</sup><http://pythonhow.com/how-a-flask-app-works>

<sup>17</sup><http://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world>

*If you use a private repository then you will have to add my account as a collaborator. On both Github and Bitbucket my account name is 'siwells'.*

Note that I won't do anything with this repo, it is just for practise for the moment, and also gives me a sense of how folk are progressing through the workbook.

**CHALLENGE #2** We'll build on the Hello Napier app over the next few weeks, but we haven't yet learned to do much more than return a string from our app (which then displays in the browser). Play around with the hello napier app that we created earlier today. What happens when you return other things in the string? Can you work out how to run some different Python code and return the result? What about getting the app to generate a different random message each time the browser page is refreshed? Finally, as you should already know HTML, investigate what happens when you put some HTML tags in the return string? Could you use this to build a more interesting page? Note that we'll use easier techniques to build HTML pages later, but I want you to get into the habit of doing the following, playing with code, experimenting, and trying out your own ideas, so that you can build your own understanding of how all the various things that we're learning work together.

Over the next few chapters we will look at a whole host of things we can do with Flask. However you should also rest assured that all of the tools we are learning will prove useful to you at some point in your career and will help you to become the best developers that you can be.



# Chapter 5

## Python Flask: Debug Mode, Errors, Routing, & Static Files

The last two weeks have got us to the point where we can build a simple ‘Hello Napier’ web app using Python and Python-Flask hosted within our Linux dev server environment and whose source code is pushed to our personal Github (or Bitbucket) repositories. 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 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. Finally we will look at serving up static files, like image files from the file system.

**VERY IMPORTANT** Chapters for this and subsequent weeks build directly on all of the skills learned in the first two chapter, Linux, SSH, and Vim from chapter 3 and Python and Python Flask from 4.

To some degree you can mix and match this and subsequent chapters and subsections to meet your needs, for example, the type of web app you want to build, but you should aim to cover all chapters by the end of the semester.

### 5.1 Flask Debug Mode

Recall that our ‘Hello Napier’ app looks like this:

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

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

```
$ flask --app hello run --host=0.0.0.0
* 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 app object and executes it. By default run() takes no arguments but we have used the `host='0.0.0.0'` argument to tell flask to allow connections from outside of the local machine, this is what enables us to access the web-app from Windows as by default we would only be able to access the web-app from within the Linux dev server. However both the app object and the run function have a number of other features that we can use. Whilst developing a new web-app one of the most important is the debug mode which we can run by setting the Flask environment variable to development, e.g.

```
$ flask --app hello --debug run --host=0.0.0.0
```

When we use this command we get slightly different output from before (because we are not running in debug mode):

```
$ flask --app hello --debug run --host=0.0.0.0
* 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 automatically restarted each time we change our code, e.g. each time we save (`:w <ENTER>`) our file after editing it in Vim.
2. Printing out debug information and a Python stack trace in the browser so that we can work out what went wrong.

If we try to access a route that causes a Python error then instead of this

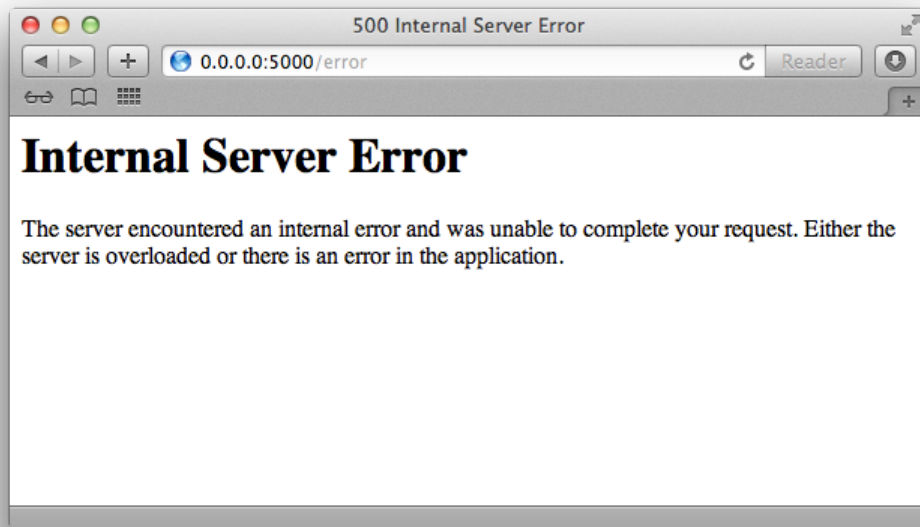


Figure 5.1: Flask internal server error

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). Here is an example but remember that the stack trace will differ greatly depending upon the type of error and the details of what went wrong so this is merely indicative:

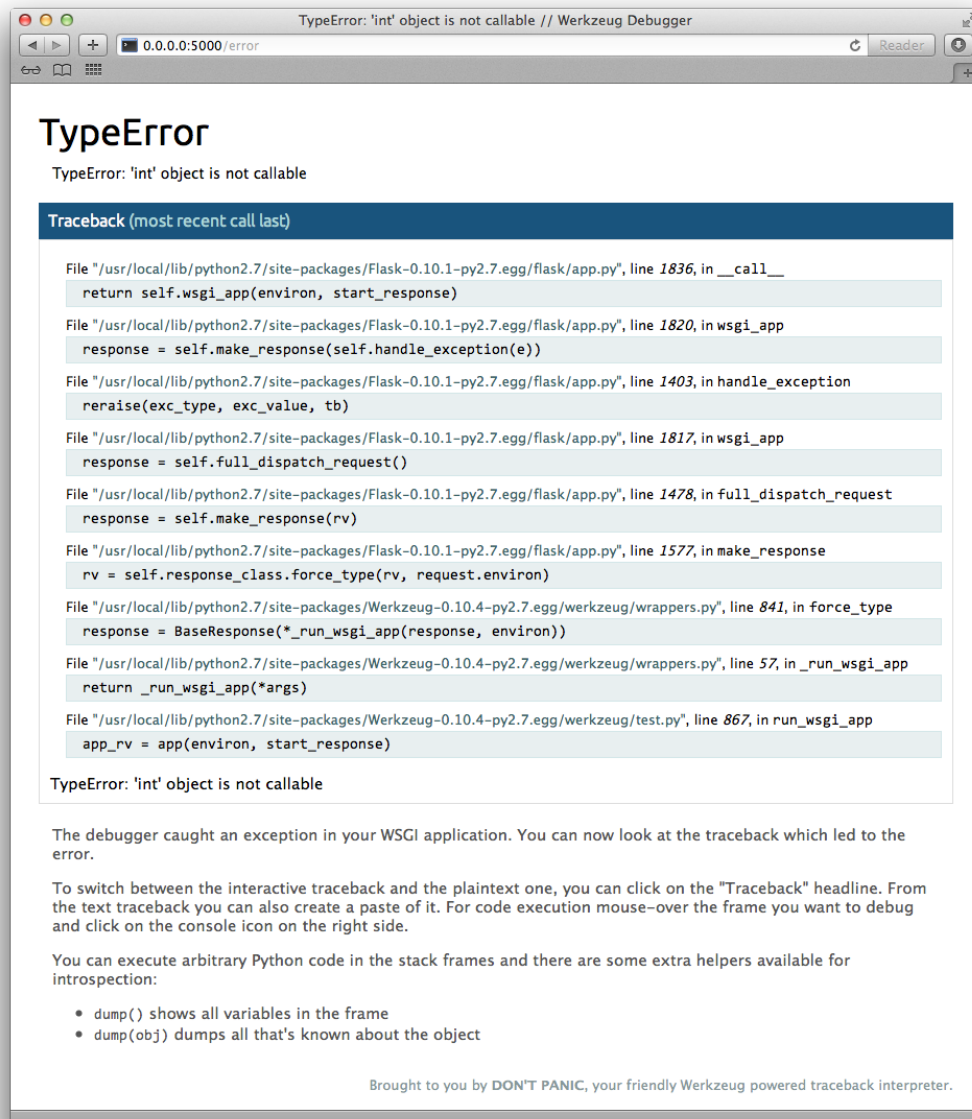


Figure 5.2: An error stack trace example

**IMPORTANT** If you ever run your web-app on a publically accessible server then you must turn debug mode off as it presents a significant security risk and can allow the execution of arbitrary code.

## 5.2 Flask Routing

Routing is what enables us to build sensible URLs and addresses for the pages of our web-app. You should consider the design of the address hierarchy for your web-app, the API, to be at least as important a consideration as the design of any content of your actual web-apps pages.

In flask, web addresses or URLs are called routes. To add more routes to your web-app you use the `app.route` decorator. You just write a new Python function then add a decorator for each one to make the function into a route. For example, in the following web-app we have 3 routes:

```
1 from flask import Flask
2 app = Flask(__name__)
3
4 @app.route("/")
5 def root():
6     return "The default, 'root' route"
7
8 @app.route("/hello/")
9 def hello():
10    return "Hello Napier!!! :D"
11
12 @app.route("/goodbye/")
13 def goodbye():
14    return "Goodbye cruel world :("
15
16 if __name__ == "__main__":
17    app.run(host='0.0.0.0', debug=True)
```

## 5.3 Flask Redirects & Errors

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 and we detected that the user wasn't logged in then we could redirect the user to a login page instead of the page that they requested, e.g.

```
1 from flask import Flask, redirect, url_for
2 app = Flask(__name__)
3
4 @app.route("/private")
5 def private():
6     # Test for user logged in failed
7     # so redirect to login URL
8     return redirect(url_for('login'))
9
10 @app.route('/login')
11 def login():
12     return "Now we would get username & password"
13
14 if __name__ == "__main__":
15     app.run(host='0.0.0.0', debug=True)
```

Note that this isn't an entire web-app, just the imports and indicative code for how a redirect could work. Obviously, to complete the scenario above we would also need code to check whether the request came from a logged in user and the login page would also need to accept and check any supplied credentials (but we will look at that type of functionality in subsequent chapters).

If we try to access a page that doesn't exist then we get a default error 404 Not Found status page like the following:

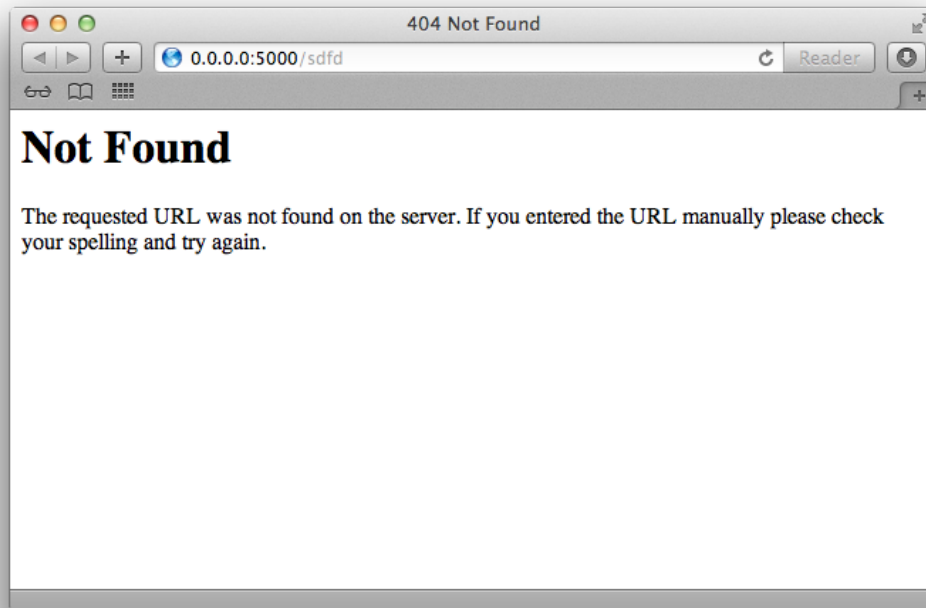


Figure 5.3: The default 404 Not Found page

However we can also provide our own error pages that better fit in with the design of our app or tell our user how to recover from the error. For example, you could provide a link to the root page or login page instead. Providing a custom HTTP error page is straightforward. Instead of the `@app.route` decorator we use the `app.errorhandler` decorator and pass it the code that we want to handle. We also add the code after the return message so that the code is returned to the browser.

```
1 from flask import Flask
2 app = Flask(__name__)
3
4 @app.route("/")
5 def hello():
6     return "Hello Napier"
7
8 @app.errorhandler(404)
9 def page_not_found(error):
10     return "Couldn't find the page you requested.", 404
11
12 if __name__ == "__main__":
13     app.run(host='0.0.0.0', debug=True)
```

Now if we visit our web-app but use an address that doesn't exist (just make something up after the `http://set09103.napier.ac.uk:5000/` bit) we should see something like this:

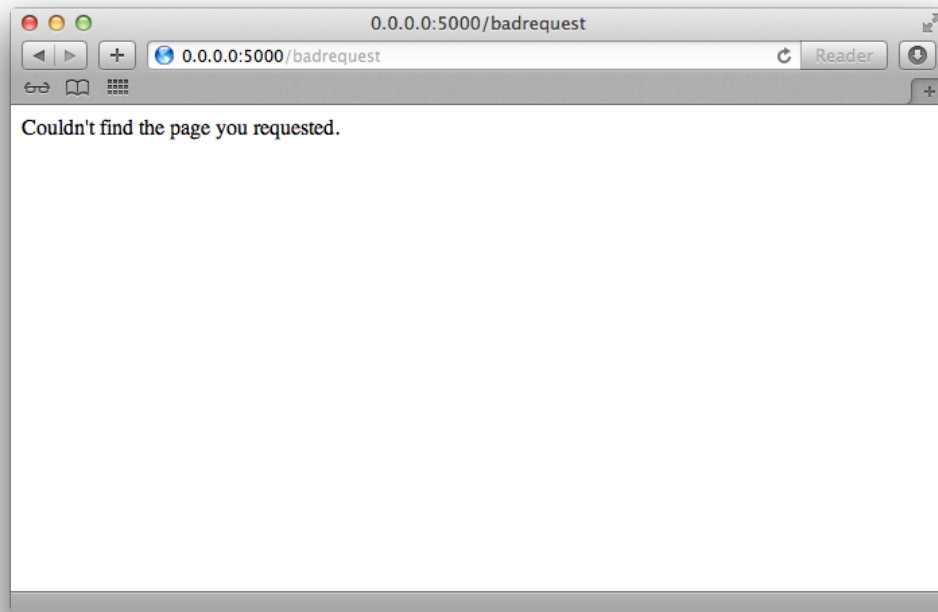


Figure 5.4: Custom 404 error page

Obviously this is not much better than the default but it does mean that we can start to build custom errors for our web-app, and once we start to look at returning HTML pages, using templates, and adding style with CSS then custom errors really become useful.

You should also be aware that there are many error codes that your web-app might conceivably respond to. Some, badly designed sites will only respond with a 404 regardless of the error that occurred, but the range of error codes actually means that you can respond with appropriate information that enables your user to make an informed decision about what to do next. Try implementing some other error codes. Because it can be difficult to force these errors to occur there is a shortcut in Flask that allows us to inject a specific type of error at any point which can be useful for testing.

```
1 from flask import Flask, abort
2 app = Flask(__name__)
3
4 @app.route("/")
5 def hello():
6     return "Hello Napier"
7
8 @app.route('/force404')
9 def force404():
10     abort(404)
11
12 @app.errorhandler(404)
13 def page_not_found(error):
14     return "Couldn't find the page you requested.", 404
15
16 if __name__ == "__main__":
17     app.run(host='0.0.0.0', debug=True)
```

Note the line that contains “abort(404)”. In this line the abort function is used which immediately causes an error to occur resulting in the corresponding errorhandler being called.

## 5.4 Flask Static Files

Even though we have seen some techniques for generating web-apps and pages dynamically from code, it is often useful to have static files, e.g. javascript, images, and CSS, that are stored in the filesystem. This enables you to incorporate useful standard web tools like, for example, JQuery, into your web-app, so you don’t have to write or generate *everything* in Python.

This means that we can generate an HTML file, to return to the client, using Jinja2 templates, but that the other artefacts of a web site, such as CSS and JS files, can be stored statically and served directly to the client. If you think about it, this makes some sense. Whilst an individual page within a site might change, to reflect the data contained within that page, many aspects remain static between and across calls. For example, the style of a site doesn’t usually change as you navigate through the site, instead it remains fairly static. In fact a hallmark of a good design is usually that it is consistent across all aspects that it applies to. So the CSS doesn’t need to be generated on the fly. Similarly images used within a site aren’t generated on the fly, icons, navigational images, or illustrative images are usually pictures that are optimised and then stored. Once stored they remain static. JS is the same, it is written and tested against design criteria, then stored as JS files, ready to be retrieved during an HTTP call. These kinds of site artefact are all static, they don’t generally need to change during a call from a client so it would probably be a waste of resources to generate them. Why not just store them statically, as files in the file systems, perhaps in a special folder where other static files are stored, until needed?

Note that this contrasts with many HTML pages themselves, particularly if the page depends upon some query from the client and might need to contain different information as a result. For example, if a page contains a table of data as a result of a database query, which in turn depends upon the parameters of the request made by the client. The data in the table needs to be assembled based on the results of the query so the page is *dynamic*. We have a different way to handle dynamic HTML using *templates* which we’ll see later in chapter 7.

To use static files, all you have to do is to create a directory called ‘*static*’ that is a sub-directory (child) of the directory in which your web-app is located. You can then place your static files, your CSS, JS, image files, &c., into this folder and Flask will automatically look there for them when an HTML file requests them<sup>12</sup>. You can, and should organise files within your static folder though. Create sub-folders so that your static files are well organised. This is particularly important as the size of your site,

---

<sup>1</sup>You can override Flask’s default placement of static files into the static folder and use some other folder name if you like using Flask’s configuration options.

<sup>2</sup>Remember that an HTML file requests other files, like CSS using the <link> tag and JS using the <script> tag



and the number of files that make up your site, grows. With small to medium sites I've found that sorting all CSS files into their own 'css' sub-folder, all JS files into their own 'js' sub-folder, and all images into their own 'img' sub-folder is sufficient to keep things organised.

Usually, if you were deploying your web-app in the wild as a public web-site then you would use a static web-server to host your static files and a web-app server, such as uWSGI, to host your dynamic flask app. This is because each is optimised for serving either static or dynamic files. A static web server takes the load of hosting and serving up the static files themselves and then delegates the dynamic pages to the appropriate web-app server. We will consider this approach later in the module<sup>3</sup>. However, during development it is sufficient to use the *static* sub-directory.

You can then retrieve any static file, for example a CSS file called *style.css*, using the Flask 'url\_for' function like so:

```
1 url_for('static', filename='style.css')
```

This function looks for the file named 'style.css' within the 'static' folder and, if found, generates the public URL for that file. In essence it translates from the internal location within the server's filesystem hierarchy, which is private to the server, to a public web address. If anyone has that address then they can retrieve the file. So if we have an image in our static folder put the result of url\_for in relation to that image into the src attribute of an img tag within an html file then that html file will retrieve and display the image. Perhaps we should try that. Let's get an image file and put it in our static folder, then afterwards we'll generate an HTML file containing an image tag in a flask route and insert the URL for our static image into that image tag.

For this to work we need an image file and we need to put that file into a /static/ subdirectory of our current flask app. First create the /static/ directory. Change directory to a folder with an active virtualenv and then create a static sub-folder within it, e.g.

```
$ mkdir static
```

Now retrieve an image file. We'll use a known good image in this example, but the file could be any image file that a browser understands. This includes JPG, PNG, and GIF files amongst others. There are many methods for getting images files onto our server but we'll use the cURL tool at the command line to retrieve a remote image from the internet and make a local copy like so:

```
$ curl -o vmask.jpg siwells.github.io/assets/images/vmask.jpg
```

Now move the image into our static sub-folder:

```
$ mv vmask.jpg static/
```

---

<sup>3</sup>Deployment, hosting, administration, & tuning of high-performance web sites is outside the scope of this module, and could alone form an entire module, however we will consider and discuss a range of tools and techniques for deploying Flask web-apps that are generally applicable to most web sites.

Now we have stored an image file, *vmask.jpg*, in the static folder we can use flask to refer to that image within our code and generated HTML. So let's generate an img link and return it to our user for display in their browser:

```
1 from flask import Flask, url_for
2 app = Flask(__name__)
3
4 @app.route("/")
5 def hello():
6     return "Hello Napier"
7
8 @app.route('/static-example/img')
9 def static_example_img():
10     start = '4</sup>.

If you now visit your VM in your browser, e.g. <http://set09103.napier.ac.uk:5000/static-example/img> replacing set09103 with your own VM ID) you will see the relative URL for the image file. It should look something like this:

---

<sup>4</sup>You can do this, and you can return *any* HTML tags this way and build up quite complex pages. However in chapter 7 we will look at how we can use templates to design how our pages look using a mixture of HTML and special coding tags to dynamically build pure HTML responses.



Figure 5.5: Displaying a static image using the *url\_for* function

Applying this to other types of static file follows a similar pattern. For example, to insert CSS into our returned page, use `url_for` to retrieve the publicly accessible address for a CSS file in your static folder, and then insert the resulting address into the `href` attribute of a `<link>` tag. Or for a JS file do the same but insert it into the `src` attribute of a `<script>` tag.

# Chapter 6

## Python Flask: Requests & Responses

In the last chapter we looked at how to do routing. That is how to execute a different function depending upon which URL the browser requested. This week we will look in more detail at the requests themselves. 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, before looking at URL encoding of arguments to a route. The, after receiving and handling request data, we will look at the responses that we can return to the client.

### 6.1 Requests

When your browser connects to a URL it is making a request. In Flask, requests are Python objects that we can access 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.

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.

```
1 print (request.method, request.path, request.form)
```

#### 6.1.1 HTTP Methods

HTTP uses various methods to move data around. The most commonly used method, and the one we have used exclusively until now is the default GET method. Look at the output from the Python Flask development server and you should see lines similar to the following:

```
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 often 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 different 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"
14
15 if __name__ == "__main__":
16     app.run(host='0.0.0.0', debug=True)
```

To test this we can browse to <http://set09103.napier.ac.uk:5000/account> and we should see the result of GET'ing the request.

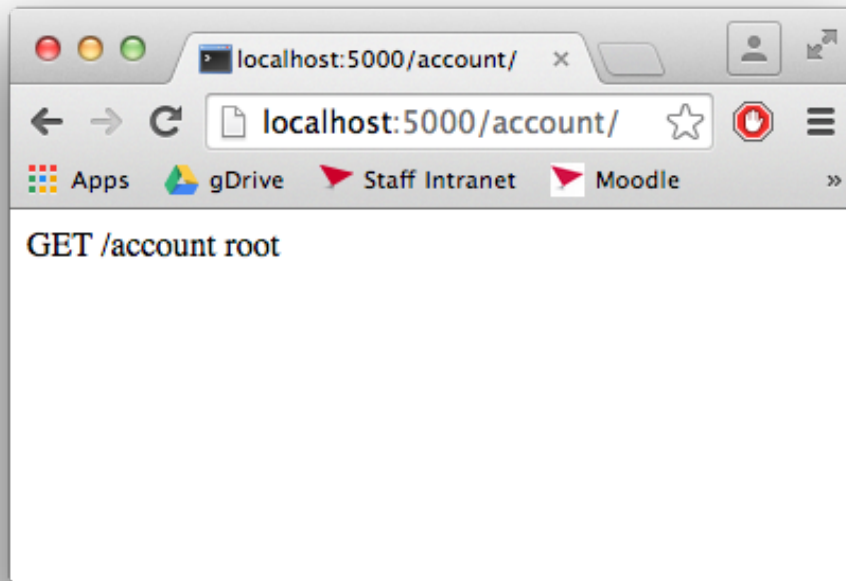


Figure 6.1: Result from GET'ing our account/ route

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 cURL for that. The easiest way is to open a new SSH window and log into the dev server then execute cURL locally on the server. We could do the same from Windows but working with the Windows command line is a pain (and we have already invested time in learning the Linux command line so we might as well continue with that. In Linux you can now just run:

```
$ curl -i -X POST http://set09103.napier.ac.uk: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 like this:

```
HTTP/1.0 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 24
Server: Werkzeug/0.10.4 Python/2.7.10
Date: Sun, 04 Oct 2015 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 6.1.1). Data transmitted in a POST or PUT request can then be accessed using the form attribute of the request object.

## 6.1.2 Request & 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.

```
1 from flask import Flask, request
2 app = Flask(__name__)
3
4 @app.route("/account/", methods=['POST', 'GET'])
5 def account():
6     if request.method == 'POST':
7         print (request.form)
8         name = request.form['name']
9         return "Hello %s" % name
10    else:
11        page = '''
12        <html><body>
13            <form action="" method="post" name="form">
14                <label for="name">Name:</label>
15                <input type="text" name="name" id="name"/>
16                <input type="submit" name="submit" id="submit"/>
17            </form>
18        </body><html>'''
19
20    return page
21
22 if __name__ == "__main__":
23     app.run(host='0.0.0.0', debug=True)
```

This should yield something similar to the following when we visit <http://set09103.napier.ac.uk:5000/account/> in the browser:

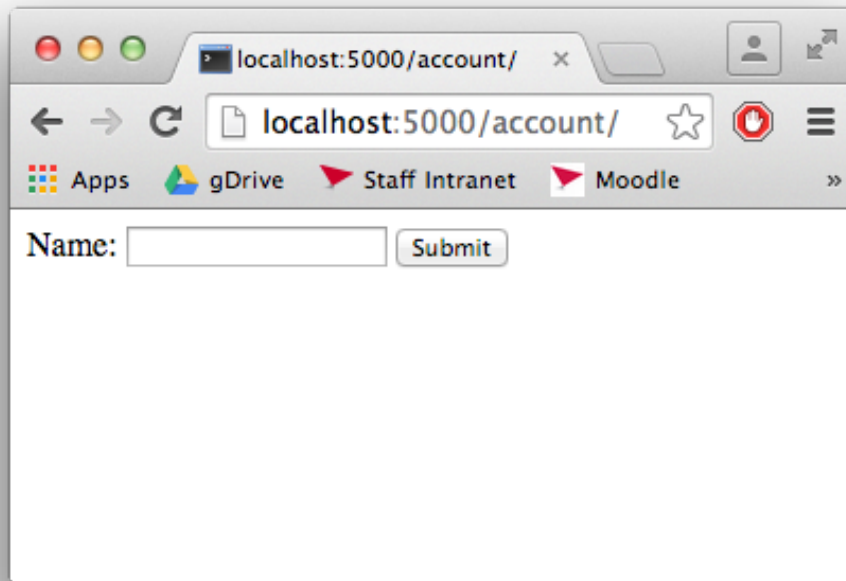


Figure 6.2: Our basic HTML form when GET'ing the account route

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 our Python function. When we enter data into the input box and click the button we should get a different page displayed as a result of the form POST'ing:



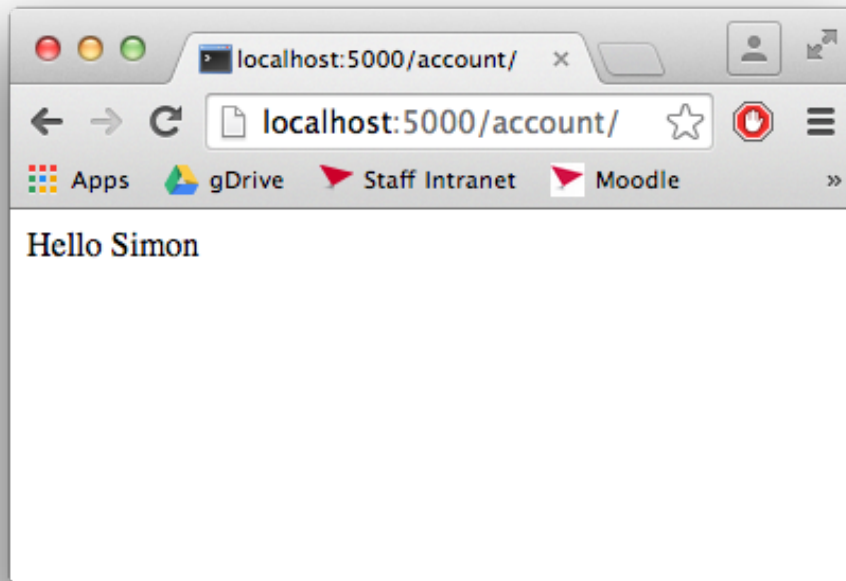


Figure 6.3: Page displayed after POST'ing the form

We will return to using POST, PUT, and other verbs in chapter 10 when we look at designing and building APIs that consume and return JSON and XML documents.

### 6.1.3 URL Variables

We can also 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. 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
7
8 if __name__ == "__main__":
9     app.run(host='0.0.0.0', debug=True)
```

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:

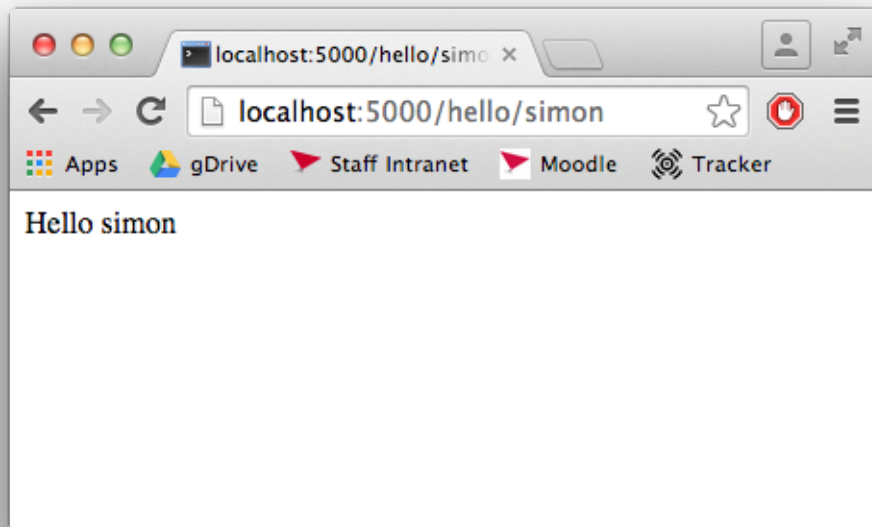


Figure 6.4: Using URL variables

By default URL variables are strings but we can 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)
7
8 if __name__ == "__main__":
9     app.run(host='0.0.0.0', debug=True)
```

With the following result:

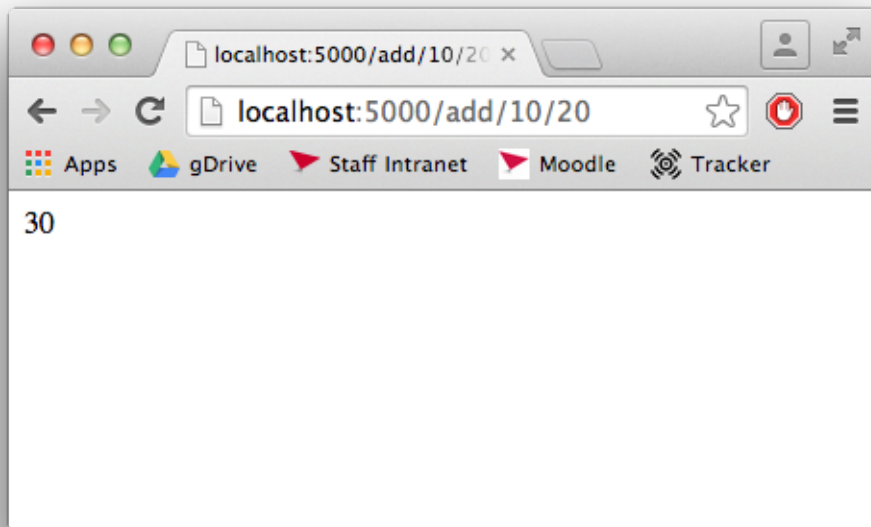


Figure 6.5: Output from using specific URL variable types

### 6.1.4 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
11
12 if __name__ == "__main__":
13     app.run(host='0.0.0.0', debug=True)
```

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

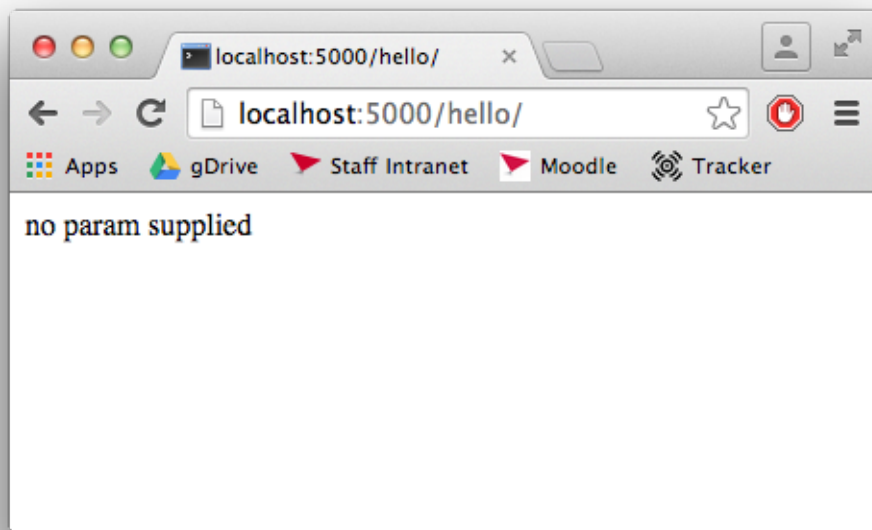


Figure 6.6: Output with no URL parameter

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

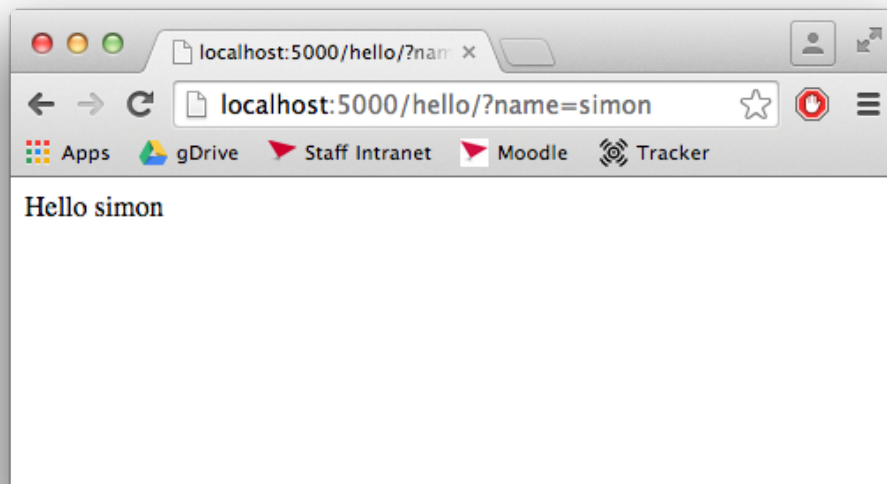


Figure 6.7: Output when supplying a ?name=simon URL parameter

### 6.1.5 Uploading Files

One final thing to consider that is related to client requests, is the matter of file uploading. We will occasionally want to enable our users to upload materials to our site and we can do this by ensuring that two things occur. The form which POSTs the data must cause the browser to transmit the file that we want to upload and our method that the route executes must also access the request and do something with the transmitted file. Otherwise the file will sit by default either in memory or in temporary storage rather than being saved for later reuse. In the following I used a PNG image file that I had available as the uploaded file:

```
1 from flask import Flask, request
2 app = Flask(__name__)
3
4 @app.route("/account/", methods=['POST', 'GET'])
5 def account():
6     if request.method == 'POST':
7         f = request.files['datafile']
8         f.save('static/uploads/upload.png')
9         return "File Uploaded"
10    else:
11        page='''
12        <html>
13        <body>
14        <form action="" method="post" name="form" enctype="multipart/form-
15            data">
16            <input type="file" name="datafile" />
17            <input type="submit" name="submit" id="submit"/>
18        </form>
19        </body>
20        </html>
        '''
```

```

21     return page, 200
22
23 if __name__ == "__main__":
24     app.run(host='0.0.0.0', debug=True)

```

Notice that the file is being saved in a sub-directory of static called 'uploads'. This makes it easier to access the uploaded file and use it within our app. Look in 'static/uploads' for the new file. Perhaps you could combine this file upload facility with the image display example that we saw in section 5.4. For example:

```

1 from flask import Flask, request, url_for
2 app = Flask(__name__)
3
4 @app.route("/display/")
5 def display():
6     return ''
8
9 @app.route("/upload/", methods=['POST', 'GET'])
10 def account():
11     if request.method == 'POST':
12         f = request.files['datafile']
13         f.save('static/uploads/file.png')
14         return "File Uploaded"
15     else:
16         page='''
17         <html>
18         <body>
19         <form action="" method="post" name="form" enctype="multipart/form-
20             data">
21             <input type="file" name="datafile" />
22             <input type="submit" name="submit" id="submit"/>
23         </form>
24         </body>
25         </html>
26         '''
27         return page, 200
28
29 if __name__ == "__main__":
30     app.run(host='0.0.0.0', debug=True)

```

Of interest here is that when we access the display method repeatedly you should see in the output from the Flask development server something similar to this:

```

10.0.2.2 - - [05/Oct/2015 17:29:14] "GET /display/ HTTP/1.1" 200
-
10.0.2.2 - - [05/Oct/2015 17:29:15] "GET /static/uploads/file.png
HTTP/1.1" 304

```

In this case we get the 304 because the image file hasn't changed (another reason we store it in our static repository. However, a final note on file uploads and static files. In a real-world deployment we usually wouldn't serve static files directly from within Flask as other HTTP servers like NGinX can do this much more reliably and efficiently. However

for development and educational purposes using the Flask static directory is an acceptable approach.

## 6.2 Responses

When we return a value from a function it is automatically turned into a valid HTML response object. 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"
7
8 if __name__ == "__main__":
9     app.run(host='0.0.0.0', debug=True)
```

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.

## 6.3 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 aspects of HTTP using Flask, so again, it can be useful 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 access 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 4.3 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<sup>1</sup>.

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 6.8.

---

<sup>1</sup>Werkzeug is a library that underpins Flask and is one of the other modules installed when you pip install Flask.



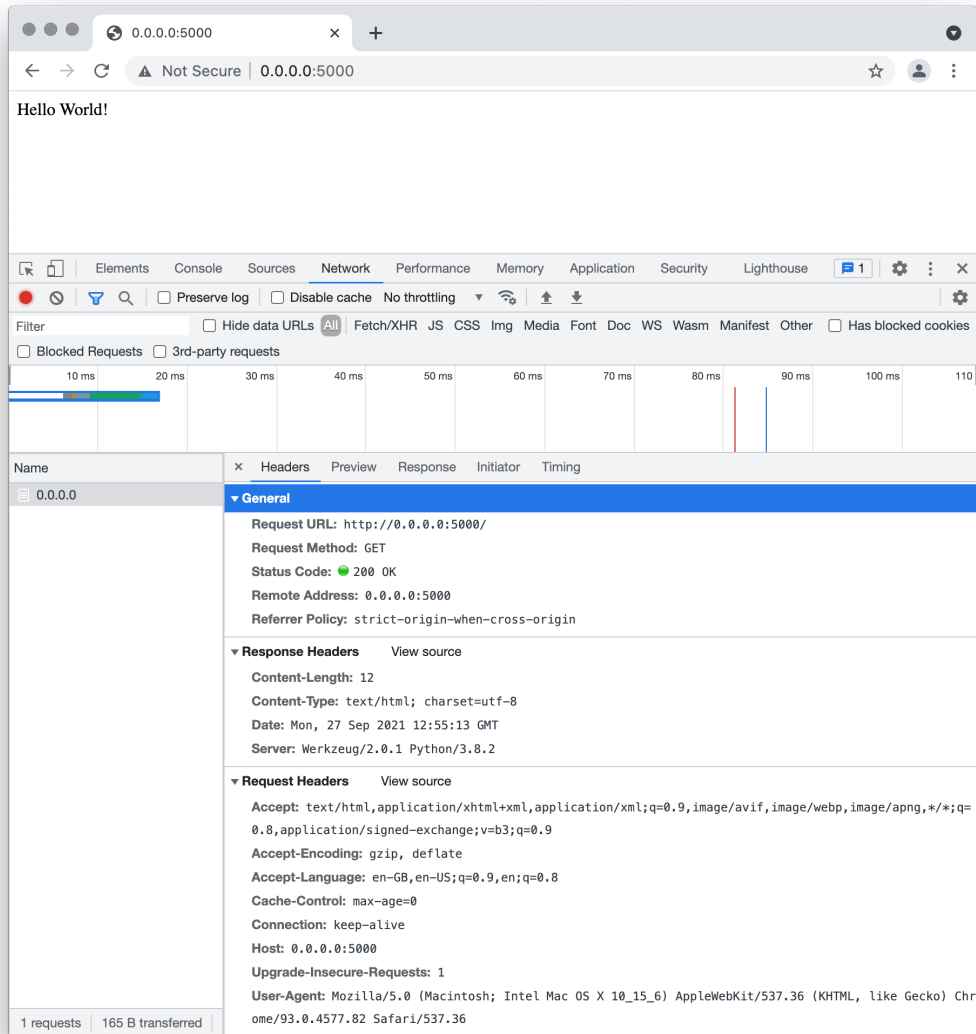


Figure 6.8: Inspecting request and response headers using the Chrome Developer Tools

# Chapter 7

## HTML Templates using Jinja2

Storing and writing our HTML code in Python as we have done in previous chapters is not a lot of fun. It is finicky and error-prone and doesn't give us much scope for doing interesting things like generating HTML pages on the fly. Luckily there is a solution for that. Using *templates* we can describe the basic layout of a page and sign-post those elements that Python can fill in with actual data. Python uses an external templating engine, called Jinja2<sup>1</sup> which is already installed on the Linux dev server alongside Python and Python-Flask.

Jinja2 is a fully-fledged Python templating engine and is not dependent upon Flask. So if you were writing another app at some point in your career that outputs HTML pages then Jinja2 is a good option. For the moment though we shall use it exclusively with Flask.

### 7.0.1 Templates & Tags

The process is simple. We supply HTML templates, in a template folder, then, in our functions we tell Flask which template to render and return to the client using the `render_template` function. So we have a couple of setup tasks to do. First, create a templates folder in the same folder as your Python Flask app, e.g.

```
$ mkdir templates
```

Now create a simple HTML template, called `hello.html` inside the templates folder, e.g.

```
$ touch templates/hello.html
```

Now open `hello.html` in Vim for editing, e.g.

```
$ vim templates/hello.html
```

Now we can put some HTML and Jinja2 tags into our template so that we can use the template from within Flask:

---

<sup>1</sup><https://palletsprojects.com/p/jinja/>

```
1 <!doctype html>
2     <h1>Hello {{ user.name }}!</h1>
3 </html>
```

What we have just done is create a template that is a mix of Jinja2 tags, indicated by the `{{`, and `}}` tags, and HTML tags, indicated by the `<` and `>` tags that we are already used to. The HTML tags are rendered as you would expect regular HTML to be treated, but we also have a variable placeholder for `user.name` which means that we can supply a variable to replace the name placeholder with. Let's use our template now in a quick Flask app:

```
1 from flask import Flask, render_template
2 app = Flask(__name__)
3
4 @app.route('/hello/<name>')
5 def hello(name=None):
6     user = {'name': name}
7     return render_template('hello.html', user=user)
8
9 if __name__ == "__main__":
10     app.run(host='0.0.0.0', debug=True)
```

Now if we call `http://set09103.napier.ac.uk:5000/hello/simon` then we should instead see this rendered template:

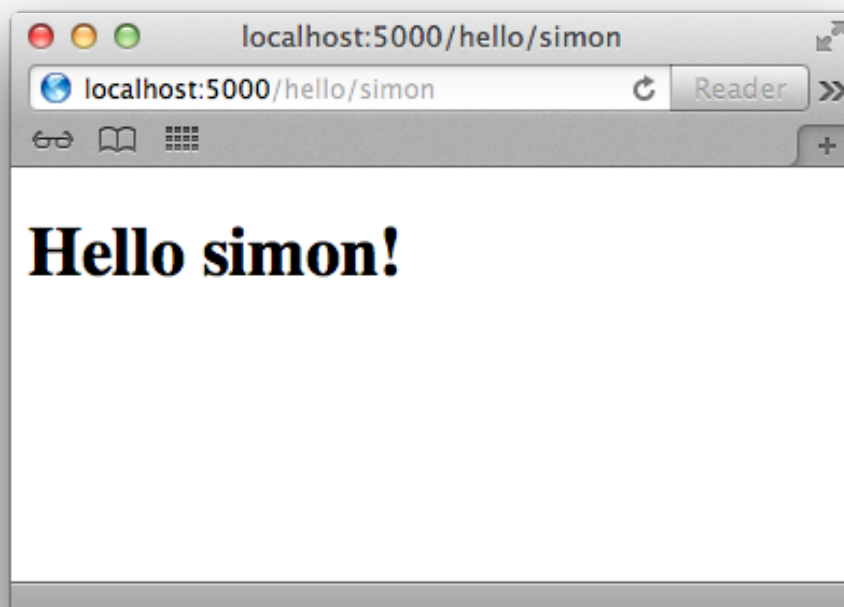


Figure 7.1: Rendered HTML with a very simple template & a single parameter

## 7.0.2 Templates with Conditional Arguments

We can also use Jinja2 templates to perform conditional behaviours, for example, rendering our HTML differently depending upon the data that is passed in. This lets us do things like personalise a page if we have a person's name or else provide a default generic page if we don't. Let's look at the template, `conditional.html`, for such a scenario (you can either create this now or get the file from the repo). `conditional.html` has the following content:

```
1 <!doctype html>
2 {% if name %}
3     <h1>Hello {{ name }}!</h1>
4 {% else %}
5     <h1>Hello from Napier!</h1>
6 {% endif %}
7 </html>
```

The Jinja2 tags cause conditional behaviour to occur; in this case they form an if...else clause, just like we have seen in many other procedural languages like Java, C, or even Python. Let's use our template now in a quick Flask app:

```
1 from flask import Flask, render_template
2 app = Flask(__name__)
3
4 @app.route('/hello/')
5 @app.route('/hello/<name>')
6 def hello(name=None):
7     return render_template('conditional.html', name=name)
8
9 if __name__ == "__main__":
10     app.run(host='0.0.0.0', debug=True)
```

There are a couple of things to notice here:

1. Notice how we have stacked up two `@app.route` calls - yes, a single function can have multiple routes defined for it, any of which can cause the function to be executed.
2. We have also used a URL variable in one of the route so that we can supply a name. Notice that the `hello` function takes a `name` argument and that it is set to `'name=None'` - this just means that we have set a default value for `name` in case the route without the URL variable is used.
3. In the `hello` function we use the `render_template` function from the Flask library which basically looks in the templates directory for a template whose name matched the one that we have supplied, `'hello.template.html'`. The function then *completes* the template, i.e. exchanging the Jinja2 tags for valid HTML tags, according to the arguments that we provide. In this case we provide the `name=name` argument, which will either have the value of `None` or else will be equal to the name that we supplied when we called the route. This argument is used to determine which path of the if...else clause to follow in the template and what value to replace any template variables with.

If we now call `http://set09103.napier.ac.uk:5000/hello` then we should see the following:

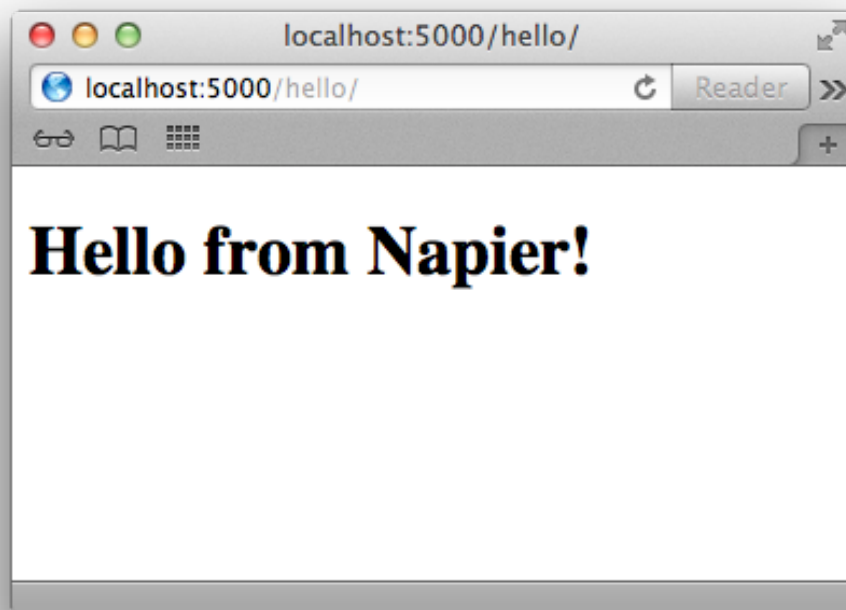


Figure 7.2: Conditional template rendering without URL arguments

But if we call `http://set09103.napier.ac.uk:5000/hello/simon` then we should instead see this rendered template:

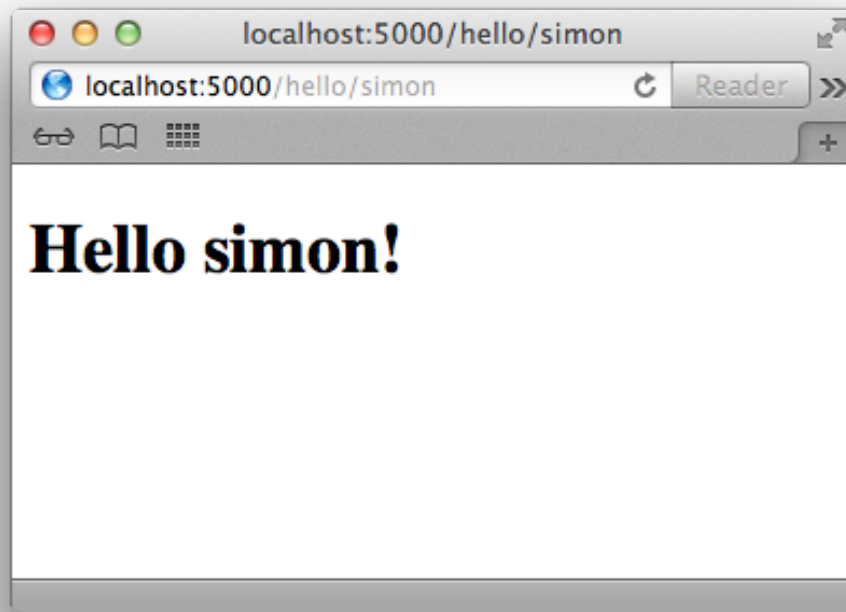


Figure 7.3: Conditional template rendering with a single URL argument

### 7.0.3 Templates & Collections

A very useful technique for generating HTML is to build a list or dictionary in Python then pass that collection into the template and cause the template to iterate over the elements of the collection. Let's look at a simple example now; here is a simple template that incorporates a Jinja2 looping construct:

```
1 <!doctype html>
2 <body>
3   <ul>
4     {% for name in names %}
5     <li>{{ name }}</li>
6     {% endfor %}
7   </ul>
8 </body>
9 </html>
```

We can now use this template in a Python Flask function as illustrated here:

```
1 from flask import Flask, render_template
2 app = Flask(__name__)
3
4 @app.route('/users/')
5 def users():
6     names = ['simon', 'thomas', 'lee', 'jamie', 'sylvester']
7     return render_template('loops.html', names=names)
8
```

```
9 if __name__ == "__main__":  
10     app.run(host='0.0.0.0', debug=True)
```

Notice that we merely constructed a Python list, a simple data structure, that is a list of names. We then passed that list into the `render_template` function for processing by the templating engine. If we now visit <http://set09103.napier.ac.uk:5000/users/> we should see that our Python list has been rendered as an unordered HTML list.

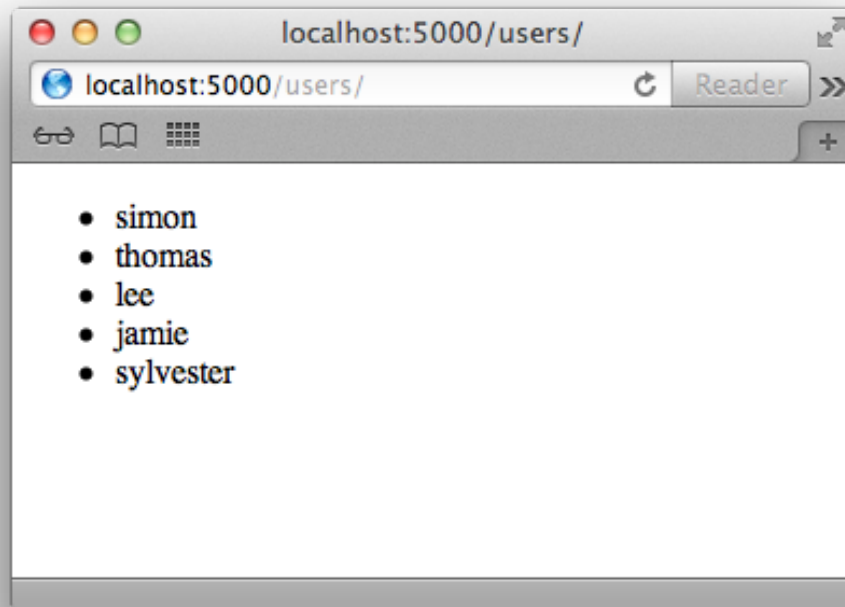


Figure 7.4: Looping over data in within a template to generate HTML

By using simple looping techniques and carefully considering the data that we pass from our Python function into a template, we can generate complex and dynamic HTML layouts.

#### 7.0.4 Template Inheritance

Because multiple templates can also be used to define headers, footers, and any other sub-part of individual pages in your app this means you can easily change and manage the look and feel of your apps. This approach is called *Template Inheritance* and means that you can, for example, define a header or menu-bar just once, e.g. in a template called `menu.html` and then include that template in every other pages which you want to display the menu. If you ever wish to add or remove a menu item then you only have to make a single edit to the menu template. Nice isn't it? This follows an established software design pattern of attempting to separate application logic separate from the presentation, markup or layout of data. Although you might ask, "what about the logic in the template?", you

are correct, there is a little bit of overlap where logic directly concerns rendering the templates but for the most part, done correctly, all of the computation of your web-app should be done in Flask and the rendering into HTML is done separately in Jinja2. This is a pretty good balance I think.

To demonstrate template inheritance we will first define a base template, then two templates that inherit from it. We will then create some routes that render the templates. So let's start with our base template:

```
1 <html>
2 <head>
3     <title>Template Inheritance Example</title>
4 </head>
5 <body>
6     <h1>Title stored in the base template</h1>
7     <h2>With a subtitle</h2>
8
9     {% block content %} {% endblock %}
10
11 </body>
12 </html>
```

For the most part this is just a normal HTML file except that it incorporates some Jinja2 tags to set out blocks that we have called 'content'. It is these blocks that are replaced within the templates that inherit from the base template. We can now inherit this base template and reuse the common elements as follows:

```
1 {% extends "base.html" %}
2 {% block content %}
3     <p>First example template. It contains some stuff</p>
4 {% endblock %}}
```

The main point to be aware of is that the *derived* template specifies that it inherits from the base template. Just to demonstrate that this works for different templates, let's create a second template that also inherits from the base template but which contains different content to the last one.

```
1 {% extends "base.html" %}
2 {% block content %}
3     <p>A second example. It's different to the other one.</p>
4 {% endblock %}}
```

We can make a Flask file which has routes that render our templates.

```
1 from flask import Flask, render_template
2 app = Flask(__name__)
3
4 @app.route('/inherits/')
5 def inherits():
6     return render_template('base.html')
7
8 @app.route('/inherits/one/')
9 def inherits_one():
```



```

10     return render_template('inherits1.html')
11
12 @app.route('/inherits/two/')
13 def inherits_two():
14     return render_template('inherits2.html')
15
16 if __name__ == ("__main__"):
17     app.run(host='0.0.0.0', debug=True)

```

Here we have three different routes. The first one `/inherits/` merely shows what the base template looks like when it is rendered without additional templates that inherit from it. In the other two routes `/inherits/one/` and `/inherits/two/` we see how the base template is modified when it is extended by two different inheriting templates. Using this approach we can build a hierarchy of page templates whilst minimising the amount of repetition by ensuring that each element that will be repeated between HTML pages is inherited from a parent template.

This is the output from our first template that inherits from `base.html`. It displays some content that comes from `inherits1.html` and some headings that are inherited from the base template.

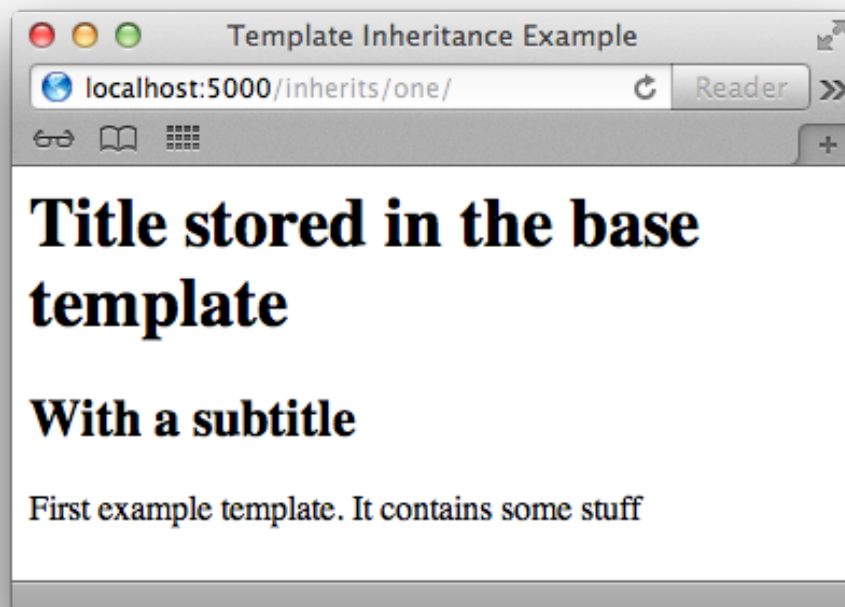


Figure 7.5: The first page that inherits from our base template

Our second page that inherits from the base template. Notice that the same headers are displayed as on the other page. However we have different content as defined by the `inherits2.html` template.

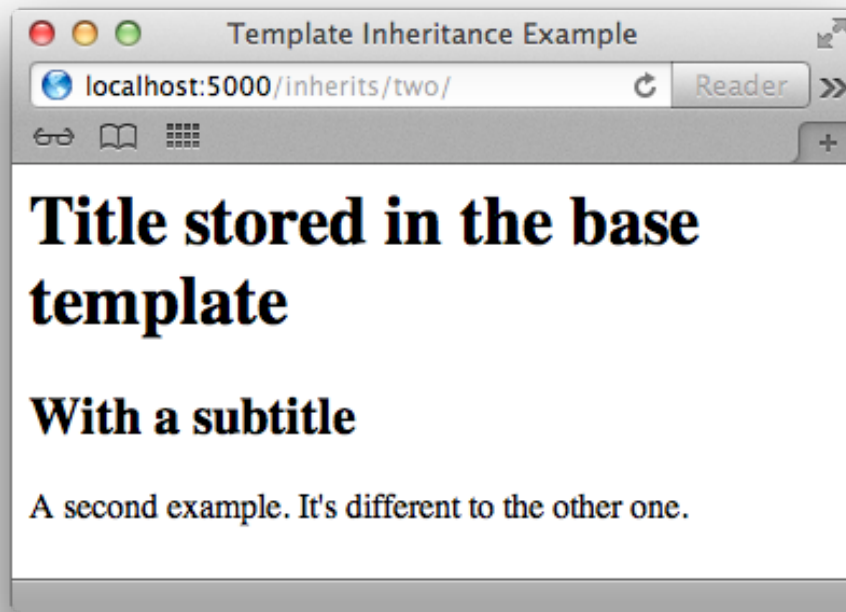


Figure 7.6: The second page that inherits from our base template

Just for completion, let's also look at what the base template looks like when rendered. We don't *have* to provide a route to it, but if we do then we can view the HTML that it generates:

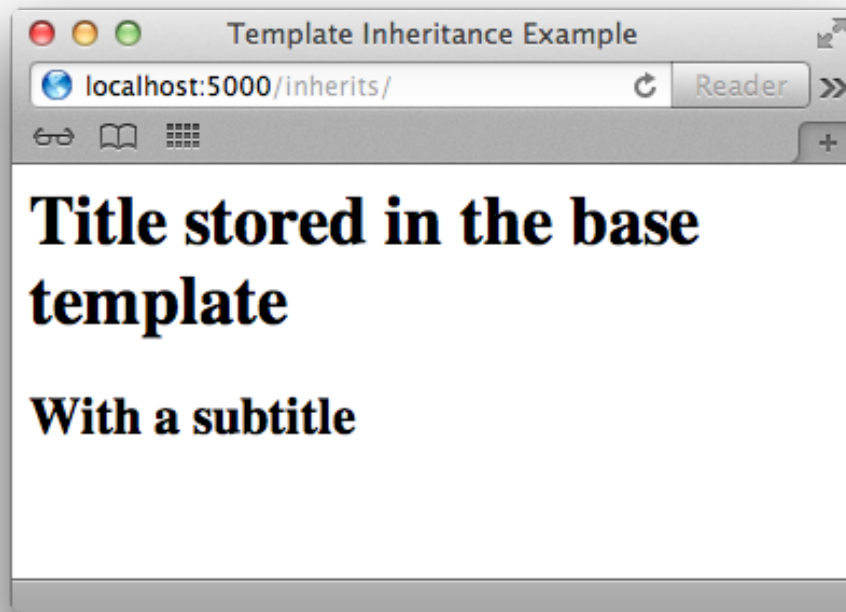


Figure 7.7: The rendered base template

You can and should use templates to describe all of the HTML pages that you want your web-apps to use. This approach is a consistent and very powerful method for generating HTML and managing the look of your web-apps.

# Chapter 8

## Flask: Configs, Sessions, Message Flashing, Logging, & Testing

### 8.1 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:

```
1 [config]
2 debug = True
3 ip_address = 0.0.0.0
4 port = 5000
5 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,

```
1 import configparser
2
3 from flask import Flask
4
5 app = Flask(__name__)
6
```

```

7 def init(app):
8     config = configparser.ConfigParser()
9     try:
10         print("INIT FUNCTION")
11         config_location = "etc/defaults.cfg"
12         config.read(config_location)
13
14         app.config['DEBUG'] = config.get("config", "debug")
15         app.config['ip_address'] = config.get("config", "ip_address")
16         app.config['port'] = config.get("config", "port")
17         app.config['url'] = config.get("config", "url")
18     except:
19         print ("Could not read configs from: ", config_location)
20
21 init(app)
22
23 @app.route('/')
24 def root():
25     return "Hello Napier from the configuration testing app"
26
27 @app.route('/config/')
28 def config():
29     s = []
30     s.append('debug:'+str(app.config['DEBUG']))
31     s.append('port:'+app.config['port'])
32     s.append('url:'+app.config['url'])
33     s.append('ip_address:'+app.config['ip_address'])
34     return ', '.join(s)
35
36 if __name__ == '__main__':
37     init(app)
38     app.run(
39         host=app.config['ip_address'],
40         port=int(app.config['port']))

```

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

## 8.2 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<sup>1</sup>.

Flask provides an interface for using ‘secured cookies’ or *sessions* from our Python code and we can consider a session to be a small data store for keys and value, a form of

---

<sup>1</sup>Unless the communication has been secured with HTTP but that is another topic

dictionary. 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<sup>2</sup>, e.g.

```
1 # Set key=value, name=simon into a session
2 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<sup>3</sup>.

```
1 try:
2     if(session['name']):
3         return str(session['name'])
4 except KeyError:
5     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.

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

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

```
1 from flask import Flask, session
2
3 app = Flask(__name__)
4 app.secret_key = 'A0Zr98j/3yX R~XHH!jmN]LWX/,?RT'
5
6 @app.route('/')
7 def index():
8     return "Root route for the sessions example"
9
10 @app.route('/session/write/<name>/')
11 def write(name=None):
12     session['name'] = name
13     return "Wrote %s into 'name' key of session" % name
14
15 @app.route('/session/read/')
16 def read():
17     try:
18         if(session['name']):
19             return str(session['name'])
20     except KeyError:
21         pass
22     return "No session variable set for 'name' key"
23
24 @app.route('/session/remove/')
25 def remove():
```

---

<sup>2</sup>If you are unsure about Python dictionaries or 'dicts' then you should do some background reading on this topic in the Python language documentation

<sup>3</sup>Again, if you are unsure about Python errors & try-except then you should do background reading on this

```

26     session.pop('name', None)
27     return "Removed key 'name' from session"
28
29 if __name__ == "__main__":
30     app.run(host='0.0.0.0', debug=True)

```

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.

```

1 $ Python
2 ...
3 >>> import os
4 >>> os.urandom(24)
5 '\xfd{H\xe5<\x95\xf9\xe3\x96.5\xd1\x010<!\xd5\xa2\xa0\x9fR"\xa1\xa8'

```

## 8.3 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

```

1 from flask import Flask, flash, redirect, request, url_for,
   render_template
2
3 app = Flask(__name__)
4 app.secret_key = 'supersecret'
5
6 @app.route('/')

```

```

7 def index():
8     return render_template('index.html')
9
10 @app.route('/login/')
11 @app.route('/login/<message>')
12 def login(message=None):
13     if (message != None):
14         flash(message)
15     else:
16         flash(u'A default message')
17     return redirect(url_for('index'))
18
19 if __name__ == "__main__":
20     app.run(host='0.0.0.0', debug=True)

```

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.

```

1 <html>
2 <body>
3 {% with messages = get_flashed_messages() %}
4     {% if messages %}
5         <ul>
6             {% for message in messages %}
7                 <li>{{ message | safe }}</li>
8             {% endfor %}
9         </ul>
10     {% endif %}
11 {% endwith %}
12 </body>
13 </html>

```

All that we have done here is add a flashed message in 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

## 8.4 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:

```

1 [config]
2 debug = True
3 ip_address = 0.0.0.0
4 port = 5000
5 url = http://127.0.0.1:5000
6 [logging]
7 name = loggingapp.log

```



```
8 location = var/  
9 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 environment 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:

```
1 import ConfigParser  
2 import logging  
3  
4 from logging.handlers import RotatingFileHandler  
5 from flask import Flask, url_for  
6  
7 app = Flask(__name__)  
8  
9 @app.route('/')  
10 def root():  
11     this_route = url_for('.root')  
12     app.logger.info("Logging a test message from "+this_route)  
13     return "Hello Napier from the configuration testing app (Now  
14         with added logging)"  
15  
16 def init(app):  
17     config = ConfigParser.ConfigParser()  
18     try:  
19         config_location = "etc/logging.cfg"  
20         config.read(config_location)  
21  
22         app.config['DEBUG'] = config.get("config", "debug")  
23         app.config['ip_address'] = config.get("config", "ip_address")  
24         app.config['port'] = config.get("config", "port")  
25         app.config['url'] = config.get("config", "url")  
26  
27         app.config['log_file'] = config.get("logging", "name")  
28         app.config['log_location'] = config.get("logging", "location")  
29         app.config['log_level'] = config.get("logging", "level")  
30     except:  
31         print("Could not read configs from: ", config_location)  
32  
33 def logs(app):
```

```

34     log_pathname = app.config['log_location'] + app.config['log_file']
35     file_handler = RotatingFileHandler(log_pathname, maxBytes=1024* 1024
    * 10 , backupCount=1024)
36     file_handler.setLevel( app.config['log_level'] )
37     formatter = logging.Formatter("%(levelname)s | %(asctime)s | %(
    module)s | %(funcName)s | %(message)s")
38     file_handler.setFormatter(formatter)
39     app.logger.setLevel( app.config['log_level'] )
40     app.logger.addHandler(file_handler)
41
42 init(app)
43 logs(app)
44
45 if __name__ == '__main__':
46     init(app)
47     logs(app)
48     app.run(
49         host=app.config['ip_address'],
50         port=int(app.config['port']))

```

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://set09103.napier.ac.uk:5000/> a new log file should be added to var/loggingapp.log and we can use an extra PuTTY window to “tail” the log so that we see each new log line appear in realtime. Tailing a log just means to show the last entries in the file and using the ‘-f’ argument to the tail command causes it to follow the log, meaning that each new line is displayed in the terminal like so:

```

$ tail -f var/loggingapp.log
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 /
^C
$

```

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.

## 8.5 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):

```
1 from flask import Flask
2 app = Flask(__name__)
3
4 @app.route('/')
5 def root():
6     return "HELLO NAPIER", 200
7
8 if __name__ == "__main__":
9     app.run(host='0.0.0.0', debug=True)
```

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 correctly, 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
11
12 if __name__ == "__main__":
13     unittest.main()
```

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.

```
$ python3 testing_test.py
.
-----

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.

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

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

```
$ python3 testing_test.py
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.

# Chapter 9

## Using Bootstrap to Add Style

We have concentrated so far on building well designed web apps that include useful features from Flask that enable us to test and configure our apps. However, we haven't spent a lot of time on making things look nice. The most we have done is use templates, in section 7, to scaffold the consistent generation of HTML for our users to read. We have also used static files, which we can use for things like Javascript, Cascading Style Sheets (CSS), icons and images, back in Section 5.4. We can further exploit the `/static/` directory to give us an easy way to add some style to our flask app as this is the correct place to store all of those elements that would be made to make our pages look nice. We should recognise however that this module is not a *web design* module, so we shall not spend time concentrating on the design of beautiful web sites, but we shall take a short cut to get a decent and consistent basic design that provide basic framework which can be further modified to produce a beautiful app.

Bootstrap is a set of CSS files and supporting Javascript that make it straightforward to add a basic style, with a selection of different layouts, to your web-app. Originally, Bootstrap was developed by Twitter as an easy way to *bootstrap* from plain HTML to a site with a basic overall design which can be enhanced and which at least looks consistent. This removes the need to initially develop HTML and CSS for layout elements like headers and footers, or menus, all you have to do is use those elements that Bootstrap has pre-defined. You can of course alter these pre-defined elements but at least you don't have to design them when you are also struggling to implement all of the actual functionality.

Because the Flask static directory hosts files that can be served up directly to the client, static is also the natural home for the Bootstrap files. Your HTML templates can then reference these CSS and Javascript files so that they are served to the client when a request is made.

As your web-app grows it is important to maintain control and organisation of your source code so I find that it is a good idea to create a hierarchy of directories within `/static/` in which to store your bootstrap files, for example,

```
static/  
static/css/  
static/font/  
static/ico/  
static/img/
```

```
static/js/
```

For the moment though we can use the directory hierarchy that Bootstrap comes with. Change directory into your static directory then download Bootstrap from <https://github.com/twbs/bootstrap/releases/download/v3.3.5/bootstrap-3.3.5-dist.zip> and unpack it, e.g.

```
$ curl -L -o bootstrap.zip https://github.com/twbs/bootstrap/
  releases/download/v3.3.5/bootstrap-3.3.5-dist.zip
$ unzip bootstrap.zip
```

Now move the bootstrap files into directory hierarchy that you created. For example, assuming you are in the static directory and you downloaded the bootstrap zip file

```
$ cp -R bootstrap-3.3.5-dist/* .
```

This should recursively copy all of the folder from within the bootstrap directory into the static directory.

We are now set up to start to use bootstrap to provide some style. But first, let's create an unstyled page, using a Jinja2 template and a simple Flask app that returns an HTML page using the template. We will then extend the template to add some bootstrap functionality to it. So, let's start with the basic template:

```
1 <html>
2 <head>
3 <title>Bootstrap Demonstation</title>
4 </head>
5 <body>
6 <h1>HELLO</h1>
7 <h2>Napier</h2>
8 <p>Demonstrating a flask app with templates for html generation and
   bootstrap
9 for a little bit of style</p>
10 </body>
11 </html>
```

Now we need a basic Flask app that uses the template:

```
1 from flask import Flask, render_template
2 app = Flask(__name__)
3
4 @app.route('/')
5 def root():
6     return render_template('base.html'), 200
7
8 if __name__ == "__main__":
9     app.run(host='0.0.0.0', debug=True)
```

Take a look at how your unstyled Flask app appears in the browser. Not very pretty is it?

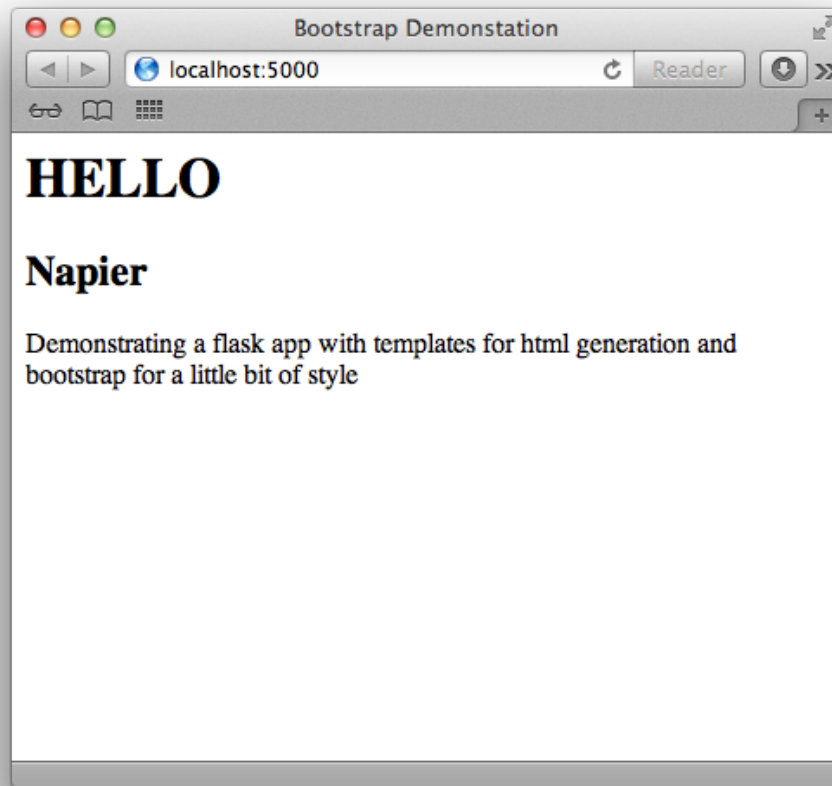


Figure 9.1: The unstyled HTML page for the Bootstrap example

Now let's use Flask to add some basic styling to that ugly old page.

```
1 <html>
2 <head>
3   <title>Bootstrap Demonstation</title>
4   <link href="{ url_for('static', filename='css/bootstrap.min.css') }}"
      rel="stylesheet" />
5 </head>
6 <body>
7 <h1>HELLO</h1>
8 <h2>Napier</h2>
9 <p>Demonstrating a flask app with templates for html generation and
    bootstrap
10 for a little bit of style</p>
11 </body>
12 </html>
```

Notice how all we have done is add a relative link, in line 4, to the Bootstrap CSS file. Also note that we used the `url_for` function to get the root of the static directory and that

all of the URL aspects are enclosed in Jinja2's double curly braces. Even, just including the bootstrap CSS has already got things looking different:

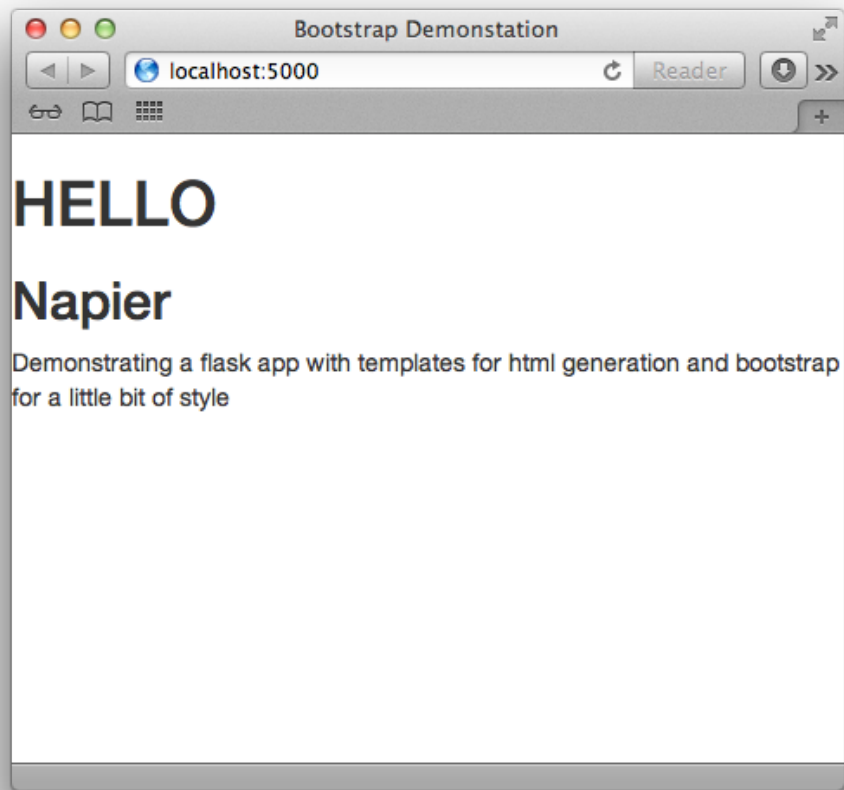


Figure 9.2: After only including the Bootstrap CSS file

But let's go a little further and implement the basic Bootstrap template:

```
1 <html>
2 <head>
3   <title>Bootstrap Demonstation</title>
4   <link href="{% url_for('static', filename='css/bootstrap.min.css') %}"
      rel="stylesheet" />
5   <style>
6     body{
7       padding-top: 50px;
8     }
9   </style>
10 </head>
11 <body>
12
13   <nav class="navbar navbar-inverse navbar-fixed-top">
14     <div class="container">
15       <div class="navbar-header">
16         <button type="button" class="navbar-toggle collapsed"
17           data-toggle="collapse" data-target="#navbar" aria-expanded="false"
18           aria-controls="navbar">
```



```

19     <span class="sr-only">Toggle navigation</span>
20     <span class="icon-bar"></span>
21     <span class="icon-bar"></span>
22     <span class="icon-bar"></span>
23     </button>
24     <a class="navbar-brand" href="#">ProjectName</a>
25 </div>
26 <div id="navbar" class="collapse navbar-collapse">
27 <ul class="nav navbar-nav">
28     <li class="active"><a href="#">Home</a></li>
29     <li><a href="#about">About</a></li>
30     <li><a href="#contact">Contact</a></li>
31 </ul>
32 </div>
33 </div>
34 </nav>
35
36 <div class="container">
37     <h1>HELLO</h1>
38     <h2>Napier</h2>
39     <p class="lead">Demonstrating a flask app with templates for
        html generation and bootstrap for a little bit of style</p>
40 </div>
41
42 <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.3/
    jquery.min.js"></script>
43 <script src="{% url_for('static', filename='js/bootstrap.min.js') %}"
    ></script>
44 </body>
45 </html>

```

There are a few things to note here. Firstly we had to add an extra inline CSS script element to push the page content below the navigation bar. Remove this line (line 6) to see what would happen otherwise. After that you can see that we used to containers, the first containing our navigation bar, and the second containing our page content. Finally we added a pair of script links, the first to JQuery<sup>1</sup> and the second to the default Bootstrap Javascript file which is in our static directory. Look on the Bootstrap website to find out what you can do with the Bootstrap Javascript, and on the JQuery homepage to see what functionality JQuery offers you.

---

<sup>1</sup><https://jquery.com/>

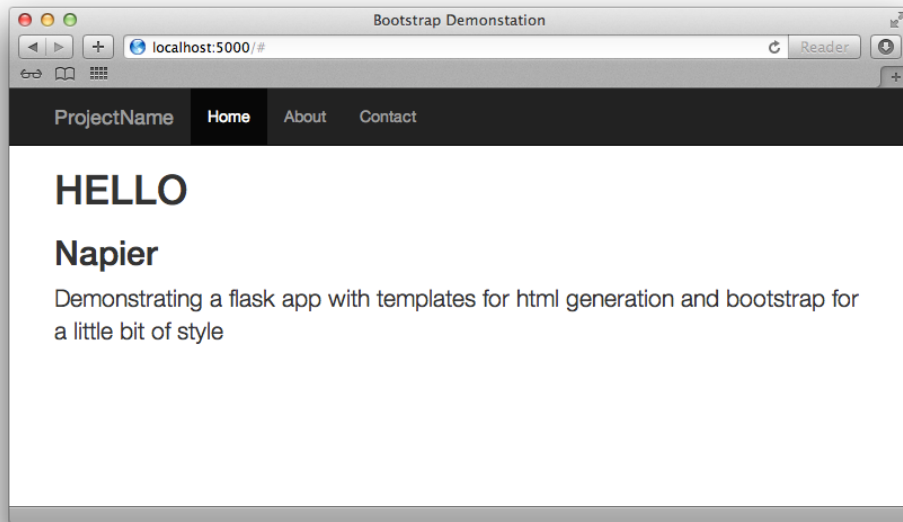


Figure 9.3: Bootstrap example with navigation bar

And this is what our page looks like in responsive mode, with a mobile style menu button:

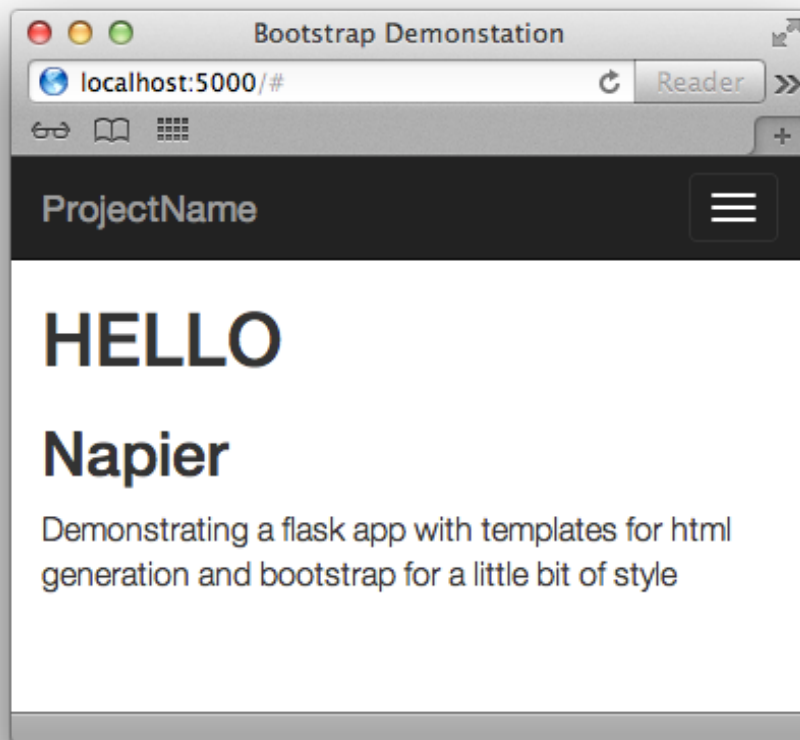


Figure 9.4: Bootstrap example in responsive mode

The bootstrap site demonstrates most of the features of bootstrap so you can see how things should look by default. It is also often useful to view the source of a given example element on the bootstrap site in order to see how it should be used. To find out about all of the things that you can do with bootstrap visit <http://getbootstrap.com/getting-started/>. Once you have got to grips with the default bootstrap there are also sites that offer additional, free, bootstrap based themes, e.g.

- <http://startbootstrap.com/>
- <http://www.bootstrapzero.com/>

Finally, if you can't find something that you like, then you can always just extend or alter the default Bootstrap themes by editing their CSS. Whatever you choose to do, Bootstrap is a good way to quickly get a not too bad looking site knocked together.

# Chapter 10

## Data Storage

Many web-apps are increasingly data driven. Whilst there are discussions to be had about the correct balance of static and dynamic functionality<sup>1</sup>, it is often the case that a database is an important part of managing the data associated with a web site.

### 10.1 Brief Introduction to SQLite3

The easiest way to get started with data storage for this module is to use SQLite3. SQLite3 is a self-contained, high-reliability, embedded, full-featured, public-domain, SQL database engine. It is also considered to be the most used database engine in the world, embedded within many apps, the default datastore on Android, and providing database functionality to support many websites.

First we need to install SQLite3. As we're on Ubuntu, which is a Debian based version of Linux, we have access to the Advanced Packing Tool (APT) which we can use to manage the installation of SQLite3. We only need to install SQLite3 once, and then it's available on the system for us to use in the future, so let's go ahead and do that now. Note that we are installing a system wide piece of software, available by default to all loggrf in users of our VM, so we have to use the sudo command to execute the installation as an admin user:

```
$ sudo apt-get install sqlite3
```

Now it's installed you can run the command line SQLite3 client by executing the following command:

```
$ sqlite3
```

This gives us a command line, a bit like the Python REPL, which we can use to directly manipulate and work with SQLite databases. I recommend exploring the SQLite3 documentation<sup>2</sup> a little and learning about manipulating SQLite databases solely within

---

<sup>1</sup>A completely static website, e.g. one that has been designed to be static or has been '*flattened*' can be **very** scalable. The equipment that powered Github pages was for many years very modest yet made tens of thousands of static web-pages for many open source projects. *NB.* Github pages also hosts the web pages for this module as well as hosting our public source code repository.

<sup>2</sup><https://sqlite.org/index.html>

the SQLite tools before proceeding to using other languages. The command line SQLite3 client looks like this:

```
$ sqlite3
SQLite version 3.28.0 2019-04-15 14:49:49
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite>
```

To exit the SQLite3 command line client and return to our terminal we can type `.exit` then hit return like so<sup>3</sup>:

```
$ sqlite3
SQLite version 3.28.0 2019-04-15 14:49:49
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite> .exit
$
```

You can also find out a little more about using this tool by typing `.help` into the tool. We can leave the SQLite tool by typing `.exit` which should return us to the regular bash shell. Remember that a database can be used independently of any given software or website so it is useful to be able to interact with our database independently of Python and Flask. To do this we could use the command line client mentioned above, or else there are a lot of third party tools for managing SQLite databases<sup>4</sup>.

However we probably really want to work with SQLite from a programming language, like Python, because this gives us a way to link our website, or really any piece of software we are writing, to an SQLite database. Using SQLite3 from Python is quite straightforward if you have worked with a relational database before. You will need to import the Python SQLite3 interface library, set a location where the SQLite database will be stored, and initialise a connection to the DB. First though you need to create a database file. I usually store database files in a “var” directory. So go ahead and create a directory called `var` then create a file within that called `sqlite3.db`.

```
$ mkdir var
$ touch var/sqlite3.db
```

We can now use that empty file we just created, `sqlite3.db`, from Python. Let’s explore this for a moment. Instead of diving right into Flask let’s just use the Python REPL and manipulate our new database a little to get a feel for interacting with it. For example, start a Python shell (by typing `python3` in the bash shell) then:

---

<sup>3</sup>Alternatively, you can send an end-of-file signal using the `<ctrl>-d` key combination which should also shut down the SQLite3 command line client. `<Ctrl>-c` and the related quit signal `<ctrl>-c` are useful tricks to have up your sleeve if you want to kill a piece of command line software that you’re stuck without resorting to the nuclear option of closing the entire window and ending the session.

<sup>4</sup>Some of these tools are graphical tools for those of you who like that kind of thing ;)

```
>>> import sqlite3
>>> DB = 'var/sqlite3.db'
>>> conn = sqlite3.connect(DB)
>>> cursor = conn.cursor()
```

In this case we have performed our imports, then created a DB object that stores the location and name of our database file, e.g. `sqlite3.db` which is stored in the relative `var` directory. We then connected to our database file and stored the connection in the variable called `conn` before retrieving a cursor that we can use to work with our database connection.

Now we have retrieved a cursor we can use it to insert and remove data from our database as well querying the data stored in it. First though, we need to set up a table. Let's create a table to store music called *albums*.

```
>>> cursor.execute(""" CREATE TABLE albums
... (title text, artist text, release_date text, publisher
... text, media_type text)
... """)
<sqlite3.Cursor object at 0x1054ec570>
>>> conn.commit()
```

Notice that we used a multi-line Python string which is wrapped in triple quotes. Now we can add some data to our database, lets add a couple of albums:

```
>>> cursor.execute('INSERT INTO albums VALUES ("Greatest Hits
...', "Roy Orbison", "30.11.1977", "SWRecords", "vinyl")')
<sqlite3.Cursor object at 0x1054ec570>
>>> conn.commit()
```

We can use a simple SQL query to see the contents of our fledgeling database as follows:

```
>>> for row in cursor.execute("SELECT rowid, * FROM albums
... ORDER BY artist"):
...     print row
...
```

We can also use SQL to carry out more complex queries but the following should give us a flavour of what we can do:

```
>>> sql = "SELECT * FROM albums WHERE artist=?"
>>> cursor.execute(sql, [("Roy Orbison")])
<sqlite3.Cursor object at 0x1054ec570>
>>> cursor.fetchall()
[(u'Greatest Hits', u'Roy Orbison', u'30.11.1977', u'
SWRecords', u'vinyl')]
```

In this case we just defined an SQL statement which we stored in the `sql` variable then we executed that statement using the `cursor.execute` function and a supplied term “Roy Orbison” to search for. As this matched a record in the database we had a record in the list that `cursor.fetchall()` returns. We can also use wildcards and the SQL “like” keyword to find close but not exact matches to our query term, e.g.

```
>>> term = "orbi"
>>> cursor.execute("SELECT * FROM albums WHERE artist LIKE
    '%{term}%'".format(term=term))
<sqlite3.Cursor object at 0x1054ec570>
>>> cursor.fetchall()[(u'Greatest Hits', u'Roy Orbison', u'
    30.11.1977', u'SWRecords', u'vinyl')]
```

For more advanced SQL queries we should look to the SQL documentation because this module isn't really about SQL query construction. For now however we can move on to look at how to integrate an SQLite3 database with Flask.

## 10.2 Using SQLite3 with Flask

Now we have seen how SQLite3 works with Python as a general data storage mechanism we can now look at how, with the addition of a few simple functions, we can provide a robust mechanism for storing our web-app's data.

We'll start with defining a schema file (`schema.sql`) that can be used to initialise our database as follows:

```
1 DROP TABLE if EXISTS albums;
2
3 CREATE TABLE albums (
4     title text,
5     artist text,
6     media_type text
7 );
```

This schema just deletes any existing table called 'albums' then create a new table called 'albums' with three text fields to store the 'title', 'artist' and 'media type' of the albums. We can now use the schema in a Flask file. Let's look at a basic Flask app (`datastore.py`) that includes a single route and some datastorage using SQLite3:

```
1 from flask import Flask, g
2 import sqlite3
3
4 app = Flask(__name__)
5 db_location = 'var/test.db'
6
7 def get_db():
8     db = getattr(g, 'db', None)
9     if db is None:
10         db = sqlite3.connect(db_location)
11         g.db = db
12     return db
```

```

13
14 @app.teardown_appcontext
15 def close_db_connection(exception):
16     db = getattr(g, 'db', None)
17     if db is not None:
18         db.close()
19
20 def init_db():
21     with app.app_context():
22         db = get_db()
23         with app.open_resource('schema.sql', mode='r') as f:
24             db.cursor().executescript(f.read())
25             db.commit()
26
27 @app.route("/")
28 def root():
29     db = get_db()
30     db.cursor().execute('insert into albums values ("American Beauty", "
31         Grateful Dead", "CD")')
32     db.commit()
33
34     page = []
35     page.append('<html><ul>')
36     sql = "SELECT rowid, * FROM albums ORDER BY artist"
37     for row in db.cursor().execute(sql):
38         page.append('<li>')
39         page.append(str(row))
40         page.append('</li>')
41
42     page.append('</ul></html>')
43     return ''.join(page)
44
45 if __name__ == "__main__":
46     app.run(host="0.0.0.0", debug=True)

```

There are four things to notice here:

1. The `init_db` function loads our schema file and initialises a new database. We could call `init_db()` each time we restart the app but this would re-initialise the data each time. Instead we will create a separate external file that calls this function whenever we need it *we shall come back to this in a moment*.
2. The `get_db` function can be called from within a route to get access to our database connection, e.g.

```
db = get_db()
```

and we can then get a cursor using

```
db.cursor()
```

3. The `close_db_connection` function is a *decorated* function that is automatically called to close the db connection whenever necessary. This is usually when a request end which causes that current context of the flask app to end.



4. Our root function which defined the `/` route does two things, it stores a new album in the database each time the route is accessed. Secondly, this function executes a simple SQL query to retrieve all the entries from the albums table then constructs a small HTML file to display those entries.

If we now run this flask app then we will get an error when we hit the `/` route so there is one last thing to do. The error occurs because we don't call the `init_db` function from anywhere. We probably only want to do this once when we deploy our web-app, otherwise we will lose all of the contents of the DB so we can create an external Python script, `init_db.py`, that will import the `init_db` function then execute it to initialise a new database.

```
1 from datastore import init_db
2 init_db()
```

We can now call our script before we start our flask app for the first time to initialise the database as follows:

```
$ python3 init_db.py
```

This should be enough SQLite3 and Flask integration to get started storing data. Obviously there is lots more to learn about the functionality that SQLite3 offers but that is an exercise for your self-directed study.

There are many data storage mechanisms and the availability of high-quality and performant datastores has increased greatly in recent years with the advent of the NoSQL approach. This approach suggests that there are **Not Only SQL** based approaches to storing data but many approaches, and the one that you choose should be based upon a sound assessment of the nature of the problem that you are tackling as well as the knowledge and experience of your development team. So, for example, there are a number of column-oriented, document oriented, graph-oriented, and key-value datastores, and many hybrids, which can be very useful when developing new web-apps. For example, when trying out various solutions to a problem it can be useful to use a schemaless datastore to hold your initial attempts at building a data model so that you do not waste time prematurely attempting to identify good database schema or relational models.

# Chapter 11

## Keeping Data safe with Encryption

Encryption is necessary to ensure that the data that we collect in our web-apps, for example, about our users, is stored safely. A good general approach is to not store any data about your users that you don't actually need for the purposes of your application.

### 11.1 Using PyCryptoDome Library for data encryption

PyCryptoDome<sup>1</sup> is the Python Cryptography Toolkit and provides a range of useful tools for encrypting and working with encrypted data in Python.

PyCryptodome is a Python library that incorporates some useful cryptography tools. We can install it as follows:

```
$ pip3 install --user pycryptodome
$
```

We can now check that this library is installed properly by using the following command:

```
$ python3 -c "import Crypto"
$
```

If you get any output other than a new prompt then the library is not available (in which case you should contact the module leader for a fix). You can now import and use PyCryptodome within your web-apps, however let's first explore what the library can do in the Python REPL, so launch python in your shell, e.g.

```
$ python3
```

We can now look at some of the things that we can use PyCryptodome to do. First we shall look at Hashing, taking input and calculating a fixed-length output called a hash-value, then we will look at Encryption, taking input and a key and producing a cypher text. The main difference between the two in practise is that hash algorithms

---

<sup>1</sup><https://www.pycryptodome.org/en/latest/>

are designed, as a rule, to be one-way or *trapdoor* functions, whilst, so long as you know the key, encryption algorithms are designed to enable you to recover the input from the cyphertext.

### 11.1.1 Hashing

The idea with Hashing is to take as input a string and calculate a fixed length output string called the *hash value*. This is a useful way to quickly determine whether two strings are the same, for example if wanting to ensure that the string that was transmitted is the one that was received. In that case we would calculate the hash value associated with the string, then transmit both the string and the hash value to the remote recipient who then recalculates the hash value from the string then compares the two hash values. if they differ then the string was altered in transit and if they are the same the string that was recieved is the one that was sent. This kind of usage is called *file integrity checking* but is also known as *checksumming* or *calculating a checksum*.

Hashing relies on various assumptions:

1. It should be very difficult (if not impossible) to guess the input string based upon the output string
2. It should be very difficult (if not impossible) to find two different input strings that have the same output (known as a *collision*)
3. It should be very difficult (if not impossible) to modify the input string without also modifying the hash value as a result.

We can hash a value, e.g. a string, using a range of hash functions but we can start with SHA256. First we will hash the value passed directly to SHA256 then we will compare it to the same value stored in a string and passed in, and a different value:

```
>>> from Crypto.Hash import SHA256
>>> SHA256.new('Hello Napier'.encode('utf-8')).hexdigest()
'8fa01d2f5f442915112a4c98d5c65c909f8b5532f01102371cf81776b91e5
>>>
>>> hello_napier = 'Hello Napier'.encode('utf-8')
>>> SHA256.new(hello_napier).hexdigest()
'8fa01d2f5f442915112a4c98d5c65c909f8b5532f01102371cf81776b91e5
>>>
>>> SHA256.new('Goodbye Napier'.encode('utf-8')).hexdigest()
'ebccfcde2209e3dff4deff67fdbae71129ea87692e40295768de48317244
>>>
```

### 11.1.2 Encryption with Block Cyphers

Block cyphers work on their input data in *blocks* of a fixed size, for example, blocks of 8 or 16 bytes in length. We'll use the Data Encryption Standard (DES)<sup>2</sup> as our example. The basic form of DES is pretty comprehensively broken as a secure encryption standard, however Triple-DES is still considered secure. There are many cryptographic algorithms available and for real-world uses it is worth investigating current best cryptographic practices and algorithms to adopt. DES works in various modes, e.g. Electronic CodeBook (ECB) mode or Cypher Feedback (CFB) mode amongst others. We will start with a demonstration of encrypting data using ECB.

```
>>> from Crypto.Cipher import DES
>>> des = DES.new(b'secret!!', DES.MODE_ECB)
>>> test = b'greeting'
>>> cipher_text = des.encrypt(test)
>>> cipher_text
b'(\xe5H\xe8\x10k\xc1['
>>>
>>> des.decrypt(cipher_text)
b'greeting'
>>>
```

Notice that we first imported the DES part of the library. We then create a new DES instance, set the mode 'DES.MODE\_ECB' and our encryption key '01234567'. We then used DES to encrypt our input string to yield a cipher text. After that we decrypted our cipher text to recover the original data.

**IMPORTANT** A limitation of DES is that the key must be exactly 8 Bytes long and the data that is encrypted must be a multiple of 8 Bytes in length. Obviously our data will seldom be a multiple of 8 Bytes so we often need to pad the data up to the next multiple of 8 Bytes.

Now let's try a different DES mode, DES CFB, to encrypt and decrypt some data. Of interest with CFB mode is that instead of each block being encrypted individually to form the cipher text each block in CFB mode is combined with the previously encrypted block. This time we shall use a slightly longer plain text to give a more interesting message, but still stick to the multiples of 8 Bytes rule. This is because we need more than one block for the combination step to work.

```
>>> from Crypto.Cipher import DES
>>> from Crypto import Random
>>> feedback_value = Random.get_random_bytes(8)
>>> des_enc = DES.new(b'secret!!', DES.MODE_CFB,
>>> feedback_value)
>>> txt = b"Hello World From Napier"
>>> cipher_txt = des_enc.encrypt(txt)
>>> cipher_txt
b"\xea\xa2\x81/\x9f\xc6\x80\xbf\xd8\xee\x81\x89M\x8f'\xe7\x9eB\xa7\xb8\xd7\xd6"
```

---

<sup>2</sup>[https://en.wikipedia.org/wiki/Data\\_Encryption\\_Standard](https://en.wikipedia.org/wiki/Data_Encryption_Standard)

```
>>> des_dec = DES.new(b'secret!!', DES.MODE_CFB,
    feedback_value)
>>> des_dec.decrypt(cipher_txt)
b'Hello World From Napier'
```

Notice that this is fairly similar to the ECB version earlier. There are two main points to notice. The first is that the DES object now takes a third argument, an 8 Byte random string and we use the Random function of the Crypto package to supply us with random data<sup>3</sup>. The main difference is that we have created two DES objects this time, one to encrypt and one to decrypt. This is necessary because of the combination step, that occurs after each block is encrypted which alters the feedback value.

### 11.1.3 Encryption with Stream Cyphers

Stream cyphers differ from Block cyphers by working on byte-by-byte on their input data, treating the data as a stream instead of discrete blocks. Stream cyphers are essentially block cyphers in which the block size is 1 Byte in length. PyCryptodome only supports two stream cyphers, ARC4 and XOR, in ECB mode. Let's look at an example of using the ARC4 algorithm:

```
>>> from Crypto.Cipher import ARC4
>>> arc4_enc = ARC4.new(b'01234567')
>>> arc4_dec = ARC4.new(b'01234567')
>>> txt = b"Hello World from Edinburgh Napier University"
>>> cyphertext = arc4_enc.encrypt(txt)
>>> cyphertext
"\xd9\xb0\x9fs)\x07r_B?\xdfz\x94\xdc\x8c='\x8b.9@\x1e\xe3@\xc68K\x1c\x18:\xad\xc3:~\xf9\x95wa\xbcB\xa0U\x08\xe9"
>>> arc4_dec.decrypt(cyphertext)
'Hello World from Edinburgh Napier University'
```

### 11.1.4 Public Key Encryption

Correctly selected and used instances of stream and block cyphers are acceptably secure and performant. However they have a fundamental flaw, in certain contexts, that is common to all security systems. People. These cyphers require both the people encrypting and the people decrypting to share the same key. Key management is a big problem, if not the major problem, that makes things like email encryption difficult for the average computer user. As a result, the average email might as well be written on a postcard.

Public-key encryption uses two keys, known as a key-pair to encrypt and decrypt data. The two keys in a key pair are created together using an algorithm that ensures that they are mathematically related to each other. One key is designated the public key and the other key is designated the private key. The public key can be shared with anyone whereas the private key is kept private and known only to the person who owns it. The

---

<sup>3</sup>It is worth noting that often the quality of encryption depends upon the quality of the randomness that is supplied to the algorithm. It is actually quite hard to get truly random numbers from a computer and even small statistical regularities in the randomness can be sufficient to break the encryption

public key is then used to encrypt some data and the private key is used to decrypt it. Because of the mathematical relation between the public and private key, data that is encrypted with one key cannot be decrypted with the same key but must be decrypted using the key's partner. One way to think about this is like using a padlock and key. You give some one a box and padlock. They put a message in the box, then they lock the box with your padlock and only you can unlock it with your key. Things are actually more complicated than this in practise but this is a good place to start from. Keys are actually very long numbers, very long prime numbers to be specific and generally, the longer the key the more secure the key. PyCryptodome provides functions for generating keys, e.g.

```
>>> from Crypto.PublicKey import RSA
>>> key = RSA.generate(2048)
>>> private_key = key.export_key()
>>> file_out = open("private.pem", "wb")
>>> file_out.write(private_key)
1674
>>> file_out.close()
>>>
>>>
>>> public_key = key.publickey().export_key()
>>> file_out = open("receiver.pem", "wb")
>>> file_out.write(public_key)
450
>>> file_out.close()
```

If you look in the folder that you ran the previous Python code in then you should notice a pair of new files. These are the public and private keys we just created. Having created a key-pair we can now use the public key to encrypt some data and then the private key to decrypt it. Let's start with using the public key to encrypt something, e.g.

```
>>> from Crypto.PublicKey import RSA
>>> from Crypto.Random import get_random_bytes
>>> from Crypto.Cipher import AES, PKCS1_OAEP
>>> data = "Soylent Green is people".encode("utf-8")
>>> file_out = open("encrypted_data.bin", "wb")
>>> recipient_key = RSA.import_key(open("receiver.pem").read())
>>> session_key = get_random_bytes(16)
>>> cipher_rsa = PKCS1_OAEP.new(recipient_key)
>>> enc_session_key = cipher_rsa.encrypt(session_key)
>>> cipher_aes = AES.new(session_key, AES.MODE_EAX)
>>> ciphertext, tag = cipher_aes.encrypt_and_digest(data)
>>> [ file_out.write(x) for x in (enc_session_key, cipher_aes
    .nonce, tag, ciphertext) ]
[256, 16, 16, 37]
>>> file_out.close()
```

Having encrypted some data with the public key (*receiver.pem*), we can now go ahead and decrypt the file we just created *encrypted\_data.bin* using our private key (*private.pem*).

```

>>> from Crypto.PublicKey import RSA
>>> from Crypto.Cipher import AES, PKCS1_OAEP
>>> file_in = open("encrypted_data.bin", "rb")
>>> private_key = RSA.import_key(open("private.pem").read())
>>> enc_session_key, nonce, tag, ciphertext =
    [ file_in.read(x) for x in (private_key.size_in_bytes(),
    16, 16, -1) ]
>>> cipher_rsa = PKCS1_OAEP.new(private_key)
>>> session_key = cipher_rsa.decrypt(enc_session_key)
>>> cipher_aes = AES.new(session_key, AES.MODE_EAX, nonce)
>>> data = cipher_aes.decrypt_and_verify(ciphertext, tag)
>>> print(data.decode("utf-8"))
Soylent Green is people

```

A couple of other useful things that we can do with the RSA algorithm are to sign and verify messages. We can sign a message using a public key and a hash algorithm in order to establish two things (1) that the message hasn't changed during transmission, and (2) that the origin, meaning the person who sent the message, can be trusted. Notice that this only holds if the key pair has not been compromised. This is the main reason why effective secure communications is difficult, because managing and sharing keys on a large scale is difficult, even for experienced computer users, and small mistakes can easily leave the entire system nearly as insecure as it would be without encryption. In fact some might say that a system that uses insecure encryption is worse, because you might believe it to be secure whereas within an un-encrypted system you already know not to trust it.

So let's sign a message, first we import a few libraries. Then we prepare a message and load in an existing key (*receiver.pem*). We then calculate a hash-value for the message and sign it using our RSA key, e.g.

```

>>> from Crypto.Signature import pkcs1_15
>>> from Crypto.Hash import SHA256
>>> from Crypto.PublicKey import RSA
>>>
>>> message = b'My namie is Inigo Montoya. You killed my
    father. Prepare to die.'
>>> key = RSA.import_key(open('private.pem').read())
>>> h = SHA256.new(message)
>>> signature = pkcs1_15.new(key).sign(h)

```

Given the plain text, the signature, and the public key (but *not* the private key) the recipient of a message can now check that the message that was sent both came from the person who owns the private key but also that the messages hasn't been maliciously altered during transmission, e.g.

```

>>> key = RSA.import_key(open('receiver.pem').read())
>>> h = SHA256.new(message)
>>> try:
...     pkcs1_15.new(key).verify(h, signature)
...     print("The signature is valid.")

```

```
... except(ValueError, TypeError):
...     print ("The signature is not valid.")
...
The signature is valid.
```

Public Key cryptography is an important tool in the fight to ensure that we can communicate in ways that are secure from eavesdropping, non-repudiable (meaning the person who sent it can be verified as such), and unaltered.

## 11.2 Using py-bcrypt Library for Password Hashing

Hash functions can be used in password management and storage. Web sites should only store the hash of a password and not the raw password itself. This way only the user knows the real password. When the user logs in, the hash of the password input is generated and compared to the hash value stored in the database. If it matches, the user is granted access<sup>4</sup>. Whilst you could use the hashes available in PyCryptodome for user passwords, this is no longer the most secure approach. Rather you should use a type of password hash that has been specifically designed for use to store password hash-values and thus mitigates many of the drawbacks of generic cryptographic hash functions in this context. Such hashes are known as *key derivation functions* and are designed to significantly slow down the process of hashing a value so that brute force attacks, where you try to calculate all possible hashes, become significantly expensive in terms of time to calculate. This contrasts with hashes used for checksumming, which obviously need to run as fast as possible.

BCrypt is the OpenBSD Blowfish password hashing algorithm that is described in a paper by Niels Provos and David Mazieres called "A Future-Adaptable Password Scheme"<sup>5</sup>. There is a Python wrapper for this algorithm called py-bcrypt<sup>6</sup> that you should use to hash password in your web-apps, at least until this algorithm is demonstrated to be flawed or a stronger system is proposed.

The BCrypt library isn't yet installed so you'll need to install it as follows:

```
$ pip3 install --user bcrypt
$
```

Now check that it's installed properly with the following command. Again, if you get any output other than the return of the prompt '\$' then you should contact the module leader for a fix:

```
$ python3 -c "import bcrypt"
$
```

---

<sup>4</sup>If you ever use a website that can help you to *recover* your password then they probably aren't hashing passwords. In this case you should probably question the security of that site and their ability to keep your data safe.

<sup>5</sup><http://www.openbsd.org/papers/bcrypt-paper.ps>

<sup>6</sup><http://www.mindrot.org/projects/py-bcrypt/>



As well as implementing a password derivation algorithm, bcrypt can also salt the supplied password so that the hashed-value is more resistant to rainbow-table based attacks. When you use the hashpw function, you can either pass in the pre-generated salt or else call gensalt() directly and the salt is stored within the generated hash-value. To hash a password we do the following:

```
>>> import bcrypt
>>> hash = bcrypt.hashpw('secretpassword'.encode('utf-8'),
    bcrypt.gensalt())
>>> hash
b'
    $2b$12$5KkwUyRUDEooMWAMBOLdnui0IJPvG2dYmP5nNyhSs1hW1xdovn0Ni
    '
```

First we import bcrypt, then we create a new hashed value using the supplied password and asking bcrypt to generate a salt. We only need to store the hash-value after that as the salt is stored within the hash-value. For example, you could now store the hash-value in your user database. When we need to verify a user's password we would do the following, assuming that you have retrieved the hash-value from your user DB as 'hash' and that the supplied password is stored in 'pass':

```
>>> hash == bcrypt.hashpw('secretpassword'.encode('utf-8'),
    hash)
True
```

and if the supplied password is incorrect we will see something like this:

```
>>> hash == bcrypt.hashpw('badpassword'.encode('utf-8'), hash
    )
False
```

This should be enough to get you started in storing your user's data securely. The rule of thumb is to keep your security arrangements as simple as possible because unwarranted complexity can hide weaknesses. You should also not implement your own security arrangements if there are already well tested alternatives. Encryption is a game for specialists and the likelihood is that you will not create anything as good as what is already available.

## 11.3 Secure login with BCrypt & Flask

Now let's combine our use of bcrypt for password hashing with some Flask decorators and Python functions so that we can protect specified routes from being accessed by users who have not supplied the correct credentials. We'll start by breaking down what we need in our Flask app. Firstly we need some routes, a default '/' route, a '/secret/' route, and a '/logout/' route. Secret is the route that we will protect from access by unauthorised users. The default route will present a login form then check the login credentials and will either reshew the login form, if authorisation fails, or else redirect to the secret route if our login is successful. Finally our logout route is used to reset the authorisation within our app as otherwise we would have to delete our cookies each time we wanted to log out.

We'll start by putting together our simple login form, which should be stored in a *templates* folder under our main python app file:

```
1 <html>
2 <head>
3 </head>
4 <body>
5
6 <form name="login_form" action="" method="post">
7
8     <input type="email" placeholder="Email" name="email">
9     <input type="password" placeholder="Password" name="password">
10    <button type="submit" class="btn" name="button" value="login">
11        Sign In
12    </button>
13
14 </form>
15
16 </body>
17 </html>
```

As you can see this merely presents two input boxes, one each for the email and password respectively, and a button. When the button is pressed the form is POSTed to the route that is listening for it, in our case the *root* route.

Now let's see the flask app itself which is stored in *login.py*

```
1 import bcrypt
2 from functools import wraps
3 from flask import Flask, redirect, render_template, request, session,
4     url_for
5
6 app = Flask(__name__)
7 app.secret_key = 'A0Zr98j/3yX R~XHH!jmN]LWX/,?RT'
8
9 valid_email = 'person@napier.ac.uk'
10 valid_pwhash = bcrypt.hashpw('secretpass'.encode('utf-8'), bcrypt.gensalt())
11
12 def check_auth(email, password):
13     if(email == valid_email and
14        valid_pwhash == bcrypt.hashpw(password.encode('utf-8'),
15        valid_pwhash)):
16         return True
17     return False
18
19 def requires_login(f):
20     @wraps(f)
21     def decorated(*args, **kwargs):
22         status = session.get('logged_in', False)
23         if not status:
24             return redirect(url_for('.root'))
25         return f(*args, **kwargs)
26     return decorated
27
28 @app.route('/logout/')
29 def logout():
```

```

28     session['logged_in'] = False
29     return redirect(url_for('.root'))
30
31 @app.route("/secret/")
32 @requires_login
33 def secret():
34     return "Secret Page"
35
36 @app.route("/", methods=['GET', 'POST'])
37 def root():
38     if request.method == 'POST':
39         user = request.form['email']
40         pw = request.form['password']
41
42         if check_auth(request.form['email'], request.form['password']):
43             session['logged_in'] = True
44             return redirect(url_for('.secret'))
45     return render_template('login.html')
46
47 if __name__ == "__main__":
48     app.run(host="0.0.0.0", debug=True)

```

Because we are keeping this login example as simple as possible, you will notice that the user credentials are hard-coded into the example using the *valid\_email* and *valid\_pwhash* variables. Notice also that *valid\_pwhash* doesn't store the password but the hashed and salted version of the password. This means that we cannot easily recover the user's password if they forget it. In a real app we would store these credentials in a user database then retrieve them as required, however we have not included that here in order to simplify the example.

We also have some utility functions, for example the *check\_auth* function that takes in an email and password, then compares them to the stored user and hashed password, returning True or False depending upon whether they match or not.

Next we have a decorator function called 'requires\_login' that we use to *decorate* any Flask route that can only be viewed by a logged in user. If the user is not logged in then they are redirected to the '/' route. The only check that we are doing within this decorator is seeing whether there is a 'logged\_in' session variable. If this check evaluates to True then the user is considered to be logged in, and not otherwise.

The 'logout' route merely removes the session variable that determines whether a user is logged in or not then redirects to the '/' route.

The 'secret' route just displays a string indicating that it is secret. Notice the additional decorator on this route. Not only do we have the '@app.route' decorator that tells Flask to treat this function as a web-app route, but we also have the '@requires\_login' route which causes the 'requires\_login' function to be run *before* the 'secret' route is executed.

Finally, the 'root' route displays our login template. If a POST is received then the *check\_auth* route is called with the supplied credentials to determine whether the user is logged in.

This is a relatively low-security way to implement a login mechanism for a Flask app. It is susceptible to various attacks, for example, a replay-attack; if an attacker were to copy a logged in user's cookie into their own browser then they would be treated as logged in, even though they had made no changes to the cookie. They wouldn't however be able to log in again as they would not have valid credentials to do so, so this attack can be mitigated to some degree by setting a timeout on the length of the user's authentication period. Another defence is to use encrypted tokens that increment for each request a user makes so that there is a strict sequence of interactions between the user and the app. If any interaction supplies an incorrect (or reused) token then the user is automatically logged out and asked to supply their credentials. Achieving a secure login that is not in any way vulnerable to attacks is a difficult task but as a developer and designer we should also evaluate the likelihood of various attacks, for example, the replay attack described above would require an attacker to gain access to your users secure cookie stored in their browser which would mean that the user is likely already heavily compromised.

However all is not lost. If we securely store our user's data and hash passwords and other sensitive data, and if we also follow a privacy-by-design methodology and only store data that is necessary for the effective functionality of our app, we can at least reduce the effects of a successful attack in terms of our user's personal data. If we are also active in logging the behaviour of our app, checking those logs, and reviewing our codebase for known vulnerabilities or attack vectors then we will also be reducing the likelihood of a successful attack occurring.

# Chapter 12

## An HTTP Server from Scratch

We’ve come a long way, and learned a lot of things, but now it feels like the best time to circle back to where we started, to simplify things, and to consider how we’d build an HTTP server, if we didn’t have Flask (or Werkzeug) to provide the foundations.

Essentially we’re going to use Python to create our own simple HTTP server. This will implement our own version of some of the simpler features of Flask, but will also implement the essential, core features of an HTTP server and demonstrate how simple it actually is to implement the most essential aspects of HTTP.

“Why do this?”, you might ask. We already have Flask<sup>1</sup>. There are many reasons to do this but the most important at this point are along the following lines:

- to see for ourselves how easy it is to implement an HTTP server,
- to see how straightforward it is to take an existing specification or standard for a protocol, like HTTP, and to create our own implementation, that is, to turn it into code that we can use in our own projects, and
- to recognise that by doing this kind of low level implementation of things we’re taking Richard Feynman’s approach to learning. He famously wrote on his own blackboard, that “What I cannot create, I do not understand”. Following this line of reasoning, if we can create an HTTP server then perhaps we can truly say that we understand that aspect of HTTP<sup>2</sup> which is perhaps the best place to bring this book to a close.
- Also, it’s fun<sup>3</sup>.

---

<sup>1</sup>... and a whole heap of other HTTP libraries in just about every language that you might use to program a computer. For example Bottle.py (<http://www.bottlepy.org/>) is another HTTP framework for Python that actually inspired the development of Flask. Bottle is simpler than Flask, contained entirely in a single Python file, and is fully commented and documented, designed to help you understand all of the parts of an HTTP framework through example

<sup>2</sup>the obvious corollary here is to consider the client side of HTTP which we don’t do in this book but which is explored in a similar chapter in the companion book “Web Technologies: Client-Side” if you’re interested in building a simple web browser

<sup>3</sup>I think. For a certain meaning of the word “fun” at any rate

So we're going to assume that we have a Web browser already, so we've got something that can make requests to our simple HTTP server and consume the response. Any browser will do so just use your favourite. By using a standard browser as our client we can demonstrate to ourselves that what we are generating in our server is what the browser expects, because we will get real web pages served up and displayed in our browser. We'll also use cURL a little bit later on, partly for practise, and partly to let us explore the features of our tiny HTTP server.

Finally, before we properly begin, and get to some code, treat this exercise akin to when you implement basic algorithms and data structure in a Computer Science class. Implement your own version so that you can prove to yourself that you understand it<sup>4</sup> and then subsequently use a pre-built and thoroughly test library or framework version in your real world code.

We're not implementing an HTTP server here that we'd necessarily use in the real world, although knowing how to write a short Python script that can serve up arbitrary data over HTTP is a *very* useful skill to have under the right circumstances. No, we're just doing this for fun and learning. When we're sure we understand things we go back to proven technologies like Flask, uWSGI, and NGinX. Just as we always use builtins and library versions of stacks, queues, and linked lists rather than rolling our own for each project.

The essence of Web serving is to listen for requests and to respond appropriately. We don't know when a request will come, so we are essentially saying that we want to wait, and listen, indefinitely, for a request to arrive.

In a moment we'll look at a very simple HTTP server implemented in Python3. Whilst I refer to it as the simplest, we could probably code golf it into something even simpler. For example, Python includes libraries that implement aspects of HTTP so we could just use those, but that feels like cheating<sup>5</sup>. Instead, my example will just use a basic networking library that's standard in Python3, but nothing else that is specific to HTTP or its implementation. This way, assuming that we have networking ability, that is, the ability to send and receive data over sockets, what do we need to do to implement the important parts of the protocol so that a Web server can usefully interact with our program?

The basic process is straightforward. Recall that HTTP is not a particularly low-level protocol, it actually relies on lower level networking to provide a networking foundation. For now we'll assume that the lowest levels of the network stack are accounted for, e.g. a physical connection<sup>6</sup>, and we'll work at the lowest level that Python needs to care for. So our process is to first create a communication path that our HTTP server can listen through. This communication path will be a network socket that will listen for communication on a specified network interface. Furthermore, the specific type of socket

---

<sup>4</sup>in the Feynmann sense

<sup>5</sup>Python3 actually has a one liner that you can use from the command line to serve up any folder that you run it in using HTTP. Really useful on occasion. Here it is: `python -m http.server`

<sup>6</sup>most likely Ethernet or Wifi

will be a TCP/IP socket. That is, a socket which uses the Internet Protocol (IP) for addressing and the Transmission Control Protocol (TCP) for transmitting and receiving data<sup>7</sup>. Once we have a TCP/IP socket created and listening, which is the bulk of the simplest example, we can then do the implementation of enough HTTP to return a valid response to a connecting Web browser.

That job of returning a valid response merely means to write plain text to our socket when a connection is recieved. The text that we write is merely three lines. The first line is the header, e.g. *HTTP/1.1 200 OK*, the next line is the blank separator between header and body, and the third line is the content of the response (which will be our HTML).

Note that in the simplest version, we cheat a little, and return the same HTTP response for any incoming connection. But later on perhaps we'll look at what the client is actually asking requesting, and vary our presponse accordingly.

So here we have our really simple HTTP server (simplest.py):

```
1 import socket
2
3 HOST, PORT = '', 8080
4
5 http_server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
6 http_server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
7 http_server.bind((HOST, PORT))
8 http_server.listen(1)
9
10 print("Serving HTTP on port {PORT} ...".format(PORT=PORT))
11
12 while True:
13     client_connection, client_address = http_server.accept()
14     request_data = client_connection.recv(1024)
15     print(request_data.decode("utf-8"))
16
17     http_response = b"""\
18 HTTP/1.1 200 OK
19
20 Hello, Napier!
21 """
22     client_connection.sendall(http_response)
23     client_connection.close()
```

Let's run it, and see it working, then we'll go through it line by line, before enhancing it some more.

Type the source code above into a file, save it as *simplest.py*, then execute it in the terminal like so:

```
$ python3 simplest.py
```

---

<sup>7</sup>TCP and IP aren't *necessary* for implementing an HTTP server but the rest of the current public Web is underpinned by both of these protocols so if we want to interact with the rest of the Web, for example, enabling existing browsers to contact our server, then we need to use the same foundational technologies.

In the terminal we should immediately see some output:

```
$ python3 http_simplest.py
Serving HTTP on port 8080 ...
```

Which is just our start up message, that we printed out, telling our user which port we're listening on. Notice that we're using port 8080, the traditional *backup* port number that Web servers listen on. It's also a useful message to remind us to specify that port number to our browser, which is actually our next step. Open a new browser tab and visit `http://localhost:8080/` and you should see something akin to the following:

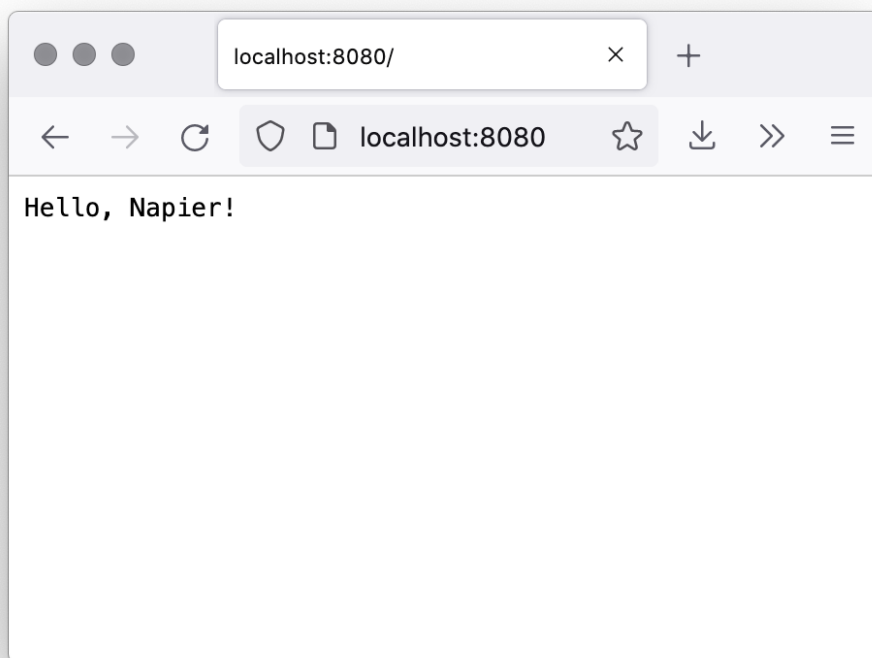


Figure 12.1: ‘Hello Napier’ from our super simple Python3 HTTP server implementation

Note that to stop our little HTTP server we need to hit `<CTRL> + c` to send a quit signal to the process we created when ran our Python script. However, before stopping it, take a look at the data that was printed to the terminal when you visited our server from your browser:

```
GET / HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv
:92.0) Gecko/20100101 Firefox/92.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,
image/webp,*/*;q=0.8
Accept-Language: en-GB,en;q=0.5
Accept-Encoding: gzip, deflate
```



```
DNT: 1
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: none
Sec-Fetch-User: ?1
Sec-GPC: 1
Cache-Control: max-age=0
```

This should be starting to look familiar... It looks like an HTTP request.

Now, as promised, let's go through that line by line, examining just what's going on. NB. We'll ignore blank lines in the formatted code. In line 1 we import the Python3 socket library which gives us an implementation of basic network sockets<sup>8</sup>:

```
import socket
```

In line 3 we then create some variables to store values for the host and port. We've specified the host as "" which is shorthand to tell our socket to listen on all reachable addresses that the machine has available to it, for example 127.0.0.1, 0.0.0.0, and localhost. We've also specified port 8080, which is the port that we want our HTTP server to listen on. Recall that 8080 is the second Web port number after port 80 but that 8080 doesn't run on a privileged port (all ports below 1024) and so doesn't require administrator privileges to run.

```
HOST, PORT = '', 8080
```

In line 5 we create our socket, naming it "http\_server" and using a call to the socket method of the socket library (socket.socket) to specify that we want our socket to be an INET socket (AF\_INET) and STREAM (SOCK\_STREAM). INET sockets are basically IPv4 sockets and STREAM sockets are TCP sockets so all we're doing is saying that the socket we want to connect should use the Internet Protocol (IP) version 4 and the it should communicate using the Transmission Control Protocol (TCP).

```
http_server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Having created our TCP/IP socket we can then set some additional options for how our socket should behave. First we want to set how our socket resuses addresses and port numbers so we can avoid "OSError: [Errno 98] Address already in use" messages. We get these when the operating system hasn't released the resources associated with the previous use of the socket. This is frustrating if, for example, we run our simple HTTP server to test it, then make a change and restart it. This bit of code hooks into lower level networking infrastructure of the OS itself, so the arguments we're passing in relate to the lower level UNIX networking specification which is why we have to pass in three arguments. The first

---

<sup>8</sup>Documentation about the socket library is available here: <https://docs.python.org/3/library/socket.html>

argument, `SOL_SOCKET`, indicates the *level* that we want our option to work at, in this case, within the socket itself, rather than within another protocol handler. The second argument, `SO_REUSEADDR`, allows us to reuse local addresses instead of waiting for them to time out or be released. The underlying `bind` program, which binds a program to a port number takes a boolean value for `REUSER_ADDR` but because a lot of underlying networking protocols are ancient in computer terms, we use booleans that are represented with the '1' or '0' value. This is because the programs were originally written in a version of C that was older than the boolean datatype is in C, so integers were used to represent them instead. In this case we are passing in the value '1' or True to say that we want to be able to reuse addresses.

```
http_server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

We now use the `bind` function to associate our newly created socket, "http\_server" with the `HOST` and `PORT` that we specified earlier. This basically fulfills the need for a single socket to be bound to a single port on the specified network interface so that it can listen for incoming messages. Remember that a given server could contain multiple network interfaces, e.g. multiple ethernet cards, wifi, and other networking mechanisms with the full range of sockets for each. This is why we need to tell the socket both which network interface and port to be associated with.

```
http_server.bind((HOST, PORT))
```

Once bound to an interface and port, i.e. localhost port 80, we explicitly tell our socket to start listening for incoming messages. This separation of binding and listening means that we can choose when to actually start receiving incoming messages separately from creating the relationship with the port. So we can reserve the port for our use, so that nothing else can clobber it first, before actively using it. Note the argument to listen, '1' which is an Integer. We use this to tell our socket how many unaccepted connections to allow from clients communicating with our socket before refusing further new connections.

```
http_server.listen(1)
```

At this point we've established our TCP/IP socket listening to port 80 on all available network interfaces so we can just print out a message to the console indicating the status of things. This is not necessary but is useful to tell ourselves and our users. Note that we are using the 'print formatted' version of the Python3 print function. This is useful if we want to output a message that contains the values associated with specific variables. Obviously whilst we write "Serving HTTP on port..." we haven't yet actually done anything that is HTTP specific, so far this has been all about setting up a TCP/IP connection that we can use for our HTTP communications.

```
print("Serving HTTP on port {PORT} ...".format(PORT=PORT))
```

As we don't want our server to consume a single message and then stop, we actually want it receive and respond to all incoming messages, we need to make our program run continuously. So we essentially use an infinite loop. As the boolean 'True' always

evaluated to true, each time we pass through the while loop it will still evaluate to true and so the loop keeps looping *ad infinitum*.

```
while True:
```

It's at this point that things start to become interesting. We've created our socket and set ourselves up to listen forever but now we just have to wait for an actual client connection. So the next piece of code runs but doesn't complete (i.e. return a value) until a client, such as a Web browser, actually connects to our socket. Once it does the accept function completes and it returns two values<sup>9</sup>, an object representing the connection itself, and a tuple containing the IP address and port number of the connecting client.

```
client_connection, client_address = http_server.accept()
```

Now that we have a successful connection, we can do something with it, like accessing the data contained in the request. The 'recv' function, along with its partner 'send', operates on a network buffer, a bit of storage provided by the operating system for storing incoming and outgoing messages. When we use recv we remove bytes from the network buffer and when we use send or sendall we add them to the buffer. Because of this we need to specify how many bytes we are dealing with for the message we are expecting to receive or transmit. Think of this a bit like communicating via postcards. You can only write a certain amount on any given postcard, unlike an envelope that you can stuff to bursting. However we can specify how large in bytes the buffer (message) should be. In this case we want to read 1024 bytes (1KB) from the network buffer.

```
request_data = client_connection.recv(1024)
```

This next function does two things as it is a function wrapped in the print function. So we are printing the result of decoding the request data that we read from the network buffer in the last line. Because historically there have been many ways for computers to encode data, we need to specify a particular encoding to use. UTF-8 is just one, now common, encoding for data. Having decoded the data we then print it to the terminal. This is where we get the information in the terminal about the incoming request. Really we should parse this and respond directly to the incoming request, for example, reading the "GET / HTTP/1.1" line and determining the type of request before responding, but for now we just dump the incoming, decoded message to the terminal.

```
print(request_data.decode('utf-8'))
```

We now prepare a response, just a string containing the information that the client expects. Note that we've assumed for the moment that our client is making an HTTP request for simplicity, but we should really check. Anyhoo, our response is a standard HTTP 200 OK response, just what our Web browser wants to see. This is followed by a single blank line and then the body content of the response, a string containing the traditional greeting of "Hello, Napier!".

---

<sup>9</sup>Remember that a nice little Python feature is that we can return multiple values from a function, really useful to assign multiple values to different variables when a function returns.

```
http_response = b"""\n
HTTP/1.1 200 OK\n
\n
Hello, Napier!\n
"""
```

Having constructed our response, which is just a string stored in the ‘http\_response’ variable at this point, we should do something with it, like send it back to the client. We do this by using the `sendall` function of the `client_connection` object.

```
client_connection.sendall(http_response)
```

Having sent our response, our HTTP request-response cycle is now complete and we can close the connection to that particular client. If there is further communication from the same client then it will cause a new `client_connection` object to be created in another iteration of our while loop. This is the basis of that stateless aspect of HTTP that we talked about earlier. Each request-response cycle is, in theory, completely independent of the previous one.

```
client_connection.close()
```

Before we elaborate on this, let’s first consider that three line response we created in the context of HTTP communication. HTTP communications are merely text sent over a communication channel. You could probably implement the protocol using people sending requests and responses on postcards or writing messages on a whiteboard if you desired<sup>10</sup>. HTTP messages, whether requests or responses, follow a pattern. There is a header and a body and these are separated by a blank line. A request from a client to a server, asking to retrieve a specific page, e.g. the page *about.html*, assuming it exists<sup>11</sup>, might look like this:

```
GET /about.html HTTP/1.1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv
:92.0) Gecko/20100101 Firefox/92.0
```

Exactly what is included in the request headers can be minimal, or can be more extensive. HTTP includes lots of additional headers that can be used to finetune the communication between client and server. The first line is the critical core though because it is in this line that we specify the HTTP verb (*GET*), the resource that the verb is targetting (*about.html*, and the version of HTTP to use (*HTTP/1.1*) when determining how to respond.

---

<sup>10</sup>but that’s much less fun than writing code to do so

<sup>11</sup>If the requested page doesn’t exist then we should really return a 404 page - but we’ll get to that later

Let's make a small change to our server. Instead of returning plain text, an http server should really be serving up hypertext, and using HTML, so let's alter the payload a little:

```
1 import socket
2
3 HOST, PORT = '', 8080
4
5 listen_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
6 listen_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
7 listen_socket.bind((HOST, PORT))
8 listen_socket.listen(1)
9 print(f'Serving HTTP on port {PORT} ...')
10 while True:
11     client_connection, client_address = listen_socket.accept()
12     request_data = client_connection.recv(1024).decode('utf-8')
13     print(request_data)
14
15     http_response = "HTTP/1.1 200 OK\n\nHello Napier!".encode()
16     client_connection.sendall(http_response)
17     client_connection.close()
18 listen_socket.close()
```

When we run this the output is ever so slightly different:

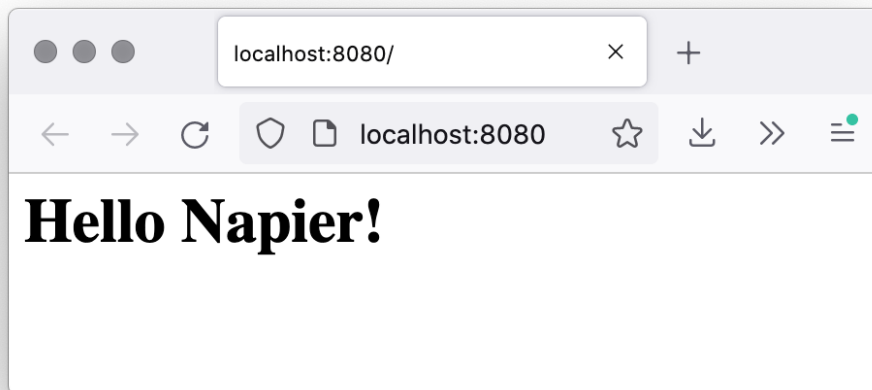


Figure 12.2: ‘Hello Napier’ as HTML from our super simple Python3 HTTP server implementation

Compare Figures 12.2 and 12.1. Try some different HTML in the payload of your http\_response to see what happens. It's also worth using view page source in your browser to investigate the differences between the plain response and responses containing HTML.

This should feel entirely satisfactory though. We're just hard-coding some HTML into our response. Wouldn't it be better if we were actually serving up an HTML file?<sup>12</sup> For

---

<sup>12</sup>It is interesting though to consider how far we could take things down the dynamic route though, of generating our page from Python on-the-fly. Likely we'd end up with something that's not a million

that we need to make some alterationsthough. How about we make our server return the content from an external file. We'll call that file `index.html` and return that for any request that arrives. First out HTML file (*index.html*) which we'll store in a sub-folder called *htmldocs*:

```
1 <html>
2 <head>
3     <title>Hello Napier</title>
4 </head>
5 <body>
6     <h1>Hello Napier!</h1>
7     <p>Welcome to the default web page (index.html) of the Napier simple
        HTTP server.</p>
8 </body>
9 </html>
```

Now we need to modify our simple HTTP server (`simplest+files.py`) to serve this HTML file whenever a request comes in:

```
1 import socket
2
3 HOST, PORT = '', 8080
4
5 listen_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
6 listen_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
7 listen_socket.bind((HOST, PORT))
8 listen_socket.listen(1)
9 print(f'Serving HTTP on port {PORT} ...')
10 while True:
11     client_connection, client_address = listen_socket.accept()
12     request_data = client_connection.recv(1024).decode('utf-8')
13     print(request_data)
14
15     infile = open('htmldocs/index.html')
16     content = infile.read()
17     infile.close()
18
19     http_response = ("HTTP/1.1 200 OK\n\n"+content).encode()
20     client_connection.sendall(http_response)
21     client_connection.close()
22 listen_socket.close()
```

All we're doing is waiting for a request, and when one arrives we're reading the contents of our `index.html` file and appending it to the HTTP response headers and new line. At this point you should notice that we're really starting to treat the headers and body content separately, generating them appropriately as needed according to the protocol. There's still a way to go yet, but we're heading in the right direction. Running this you should get something like the following displayed in your Browser:

---

miles away from Flask.

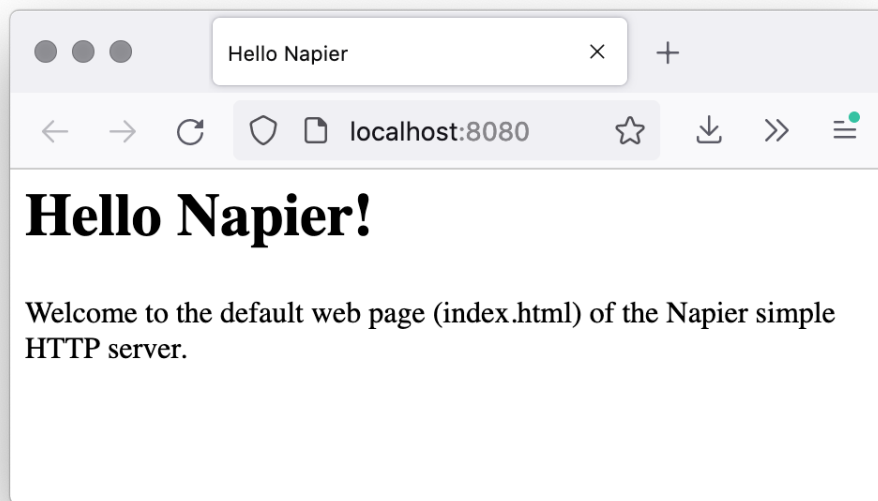


Figure 12.3: Serving index.html from our super simple Python3 HTTP server implementation

You can alter the contents of index.html as much as you like and that file content at least will be returned to the client. However we can't add in external static files for CSS or JS (yet) because our HTTP server doesn't yet support additional calls to retrieve them. Note though that you can inline CSS and JS and the browser should display and execute your page as normal.

We should probably wrap things up by clarifying that this approach isn't how you should implement your own real-world Website. It is only for *pedagogical* purposes, that is, just a learning exercise. In the real world we want more features, increased management opportunities for deployed sites, and reliability and robustness. Might this simplest example provide a starting place for a project to build the next generation of Python3 based HTTP server software? Perhaps. More likely though that under most circumstances<sup>13</sup> you should adopt something is already in existence, working well, used by, and bug-fixed by thousands. With these criteria there are many Web frameworks out there, ready to be used. Python3 and Flask are not a bad place to start from though. After that, your journey will go where it must in order to solve the problems you've set of yourself.

---

<sup>13</sup>unless you really have a burning desire to create a new HTTP server.

# Part III

## Appendices



# Appendix A

## Cribsheets

These cribsheets are useful for collecting together lots of new syntax but are no substitute for your own notes (and practise. Stuff you know is much better than stuff you can look up). Either way, as you learn new stuff you should expand these cribsheets with extra commands that you find useful.

### A.1 Linux

#### A.1.1 Some useful aliases

- ~ An alias that means your home directory within the filesystem hierarchy. On the Linux dev server, for the user 40001111 this would expand to /home/40001111, e.g. If you are lost in the filesystem then the following will always take you home:

```
$ cd ~
```

- . An alias that means the current directory. So if you want to explicitly affect something in the current directory you use this. It can help to avoid ambiguity. For example, when running a script, we specify that we want to run the one in the current folder rather than another one on the system executable path:

```
$ ./runme.sh
```

- .. An alias that means the parent of the current directory. So if you want to move from the current directory to it's parent then you can use this:

```
$ cd ..
```

- An alias that means the *previous* directory. This is really useful to help you jump between two directories in different parts of the filesystem with something like:

```
$ cd -
```

#### A.1.2 Some useful commands

**cat** *filename* Display the contents of the file *filename*

**cd** Change to your home directory /home/tc or ~/home

**cd ..** Change directory to the parent of the current directory

**cd *directoryname*** Change directory to the named directory

**ls** List the names of the files in the current directory

**ls *directoryname*** List the contents of the named directory

**mkdir *directoryname*** Create a new directory in the current directory

**mv *source destination*** Move a file (or directory) to a new location

**mv *old new*** Rename a file (or directory)<sup>1</sup>

**pwd** Display the path to the current directory in the filesystem hierarchy, e.g. show you where you are relative to the root

**rm *filename*** Delete the named file

**rm -rf *directoryname*** Will delete the named directory and all of its contents

**touch *filename*** Will create a new file called filename

## A.2 Vim

**\$ vim** - Shell command to start a new unnamed empty document in Vim

**\$ vim filename.txt** - Shell command to open 'filename.txt' in Vim. If it exists then the file will be opened, otherwise an empty file will be opened for editing that will be saved as 'filename.txt' when you use the (w)rite command

**<ESC>** - Enter command Mode

**<ESC>i<ENTER>** Enter (i)nsert edit mode

The following are a core set of Vim commands that are all used whilst in Command mode, e.g. after typing <ESC>

**:q<ENTER>** - (q)uit

**:q!<ENTER>** - (q)uit and discard any changes

**:w<ENTER>** - (w)rite changes to file

**:wq<ENTER>** - (w)rite changes to file then (q)uit

**:e *filename* <ENTER>** - Open file *filename* in Vim for editing

**dd<ENTER>** - Delete the entire line that the cursor is on

---

<sup>1</sup>Yes, these two instances of mv are very similar. This is because under the surface mv works by altering a pathname which means that moving a file or folder around the file system is technically the same as renaming a file. When a move happens it is not actually moving a file to another location on disk, just renaming where it is reported as located.

**x<ENTER>** - Delete the character that the cursor is on

**j** - Move the cursor up one line (NB. You can also use the ‘up’ arrow key)

**k** - Move the cursor down one line (NB. You can also use the ‘down’ arrow key)

**l** - Move the cursor right one character (NB. You can also use the ‘right’ arrow key)

**h** - Move the cursor left one character (NB. You can also use the ‘left’ arrow key)

**gg** - Go to start of file

**G** - Go to end of file

**\$** - Move cursor to the end of the current line

**0** - Move cursor to start of current line (NB. That's a zero)

**<CTRL>e** - Scroll up

**<CTRL>y** - Scroll down

**<CTRL>b** - Page Up

**<CTRL>f** - Page Down

**/search-term** - Search forward for ‘search-term’ in the current file (Use ‘n’ for (n)ext match in current direction and (N) for next match in opposite direction)

**?search-term** - Search backward for ‘search-term’ in the current file (Use ‘n’ for (n)ext match in current direction and (N) for next match in opposite direction)

**u** - Undo the last command

**.** - Repeat the last command

Vim has many more commands and many ways in which individual commands can be composed into more complex composite commands. We’ve seen above a core set of essential commands, now we’ll have a smattering of interesting further commands that are useful when editing and will give you a flavour of what Vim has to offer:

**J** - Combine (“join”) next line with this one

**nG** - Move cursor to line n, e.g. 1G will take you to the first line of the file

**ma** - Mark current position

**d’a** - Delete everything from the marked position to here

**‘a** - Go back to the marked position]

**:s/s1/s2** - Replace (“substitute”) (the first) s1 in this line by s2

**:set number** Turn on line numbering. Add this to your .vimrc file to have numbers on by default. Use *:set nonumber* to turn line numbers off again.

# Appendix B

## Annotated Code Examples

### B.1 Python3 Flask ‘Hello Napier’

An annotated walk through the code from `hello.py` that we saw in section 4.2.

```
1 from flask import Flask
2 app = Flask(__name__)
3
4 @app.route("/")
5 def hello():
6     return "Hello Napier!"
7
8 if __name__ == "__main__":
9     app.run(host='0.0.0.0')
```

**Line 1** *from flask import Flask*

Import the Flask class from the flask library. The library contains pre-written code and utilities that are useful when writing a web-app. In this case an instance of the Flask class will be our WSGI application.

**Line 2** *app = Flask(\_\_name\_\_)*

Create an instance of the Flask class. The argument ‘`__name__`’ is the name of the flask applications module. This is used to help flask to find resources relative to the Python module such as static web resources like image files, templates, or CSS. We also create a variable, ‘`app`’, that references the newly instantiated Flask class so that we can use it later.

**Line 4** *@app.route("/")*

Lines that start with `@` in Python are decorators. In this case we use the `route()` decorator to tell Flask which URL should trigger the function that `route()` decorates, e.g. when a browser hits the root of the url, ‘`/`’ then the `hello()` function is run. We use `route()` decorators in flask to build up our HTTP API that a browser can retrieve.

**Line 5** *def hello():*

This defines a function called ‘`hello()`’. `hello()` is executed whenever someone requests the root url.

**Line 6** *return "Hello Napier!"*

All our `hello()` does is to return the string “Hello Napier”. It is this string that

is displayed in the browser. We could instead return some HTML for a richer experience but plain text is sufficient for now.

**Line 8** *if \_\_name\_\_ == "\_\_main\_\_":*

This is used to control how the Python module and the flask app server is run. We only want to use `app.run()` if this script is executed from the Python interpreter, e.g. by calling `$python hello.py`. If we were to use an app server instead then the `app.run()` would be performed differently.

**Line 9** *app.run(host='0.0.0.0')*

Calls the `run()` function of the Flask app class instance to start our development server running using this app as the web app. This line also tells the app to run on a network interface that is accessible from an external address, e.g. from the Windows machine that is running your SSH connection, otherwise our app would only be accessible within the dev server and we don't have a graphical browser installed there.

# Appendix C

## Additional Miscellaneous (but useful) Tools

### C.1 cURL

The cURL tool is a command that you can use to interact with remote HTTP APIs. It can function purely as a download tool in the terminal, for example, in Section 5.4 we used cURL to retrieve an image file for use in our Flask static file demonstration, e.g.

```
$ curl -L siwells.github.io/assets/images/vmask.jpg -o vmask.jpg
```

By default cURL will perform an HTTP GET to the URL that we provide, However we can also specify the HTTP verb that we want to use as an additional arguments to the tool, e.g.

```
$ curl -X POST -d "firstName=jebediah" http://dummy.com/persons/person
```

Notice the use of ‘-X POST’ to specify the verb (which could be any verb defined by the HTTP standard, e.g. GET, POST, PUT, DELETE, HEAD, OPTIONS, &c.). Some verbs also expect a payload so we have also included one using the ‘-d’ argument and providing a key and value. This could actually be a whole JSON document or separate file that is used as the payload but for now we’ll keep things simple.

We can cause cURL to print extra information by using ‘-i’ option and can also use ‘-H’ to specify the accept header for the request. In the following case indicating that the payload is of type ‘application/json’

```
$ curl -i -H "Accept: application/json" -X POST -d "firstName=jebediah" http://dummy.com/persons/person
```

### C.2 Pip

Pip is a package management tool for Python libraries and is really easy to use. You can use pip to install Python libraries into your own user account on the server, e.g.

```
$ pip3 install --user PyCryptodome
```

What this does is enter the *superuser* (SU) role, then it installs pip. The superuser role has more powers than a regular user so once we have installed pip we want to exit the superuser role back to the normal user role that we logged in as. The change in role is indicated by the change in prompt that we see from *\$* to *#*. We can now use Pip to install whichever Python libraries we want to use as a normal user (not as a superuser).

First, let's list the installed Python libraries

```
$ pip3 freeze
```

It is useful to save the output of pip freeze to a file that you store in your Python app's Git repository so that you can easily reinstall all of the necessary libraries if you need to install your app elsewhere. You can do this as follows:

```
$ pip3 freeze > requirements.txt
```

Which will cause the output to be saved in a file called requirements.txt<sup>1</sup> Have a look at the contents of requirements.txt using either Vim or the cat command in the shell, e.g.

```
$ cat requirements.txt
```

Compare the output to that of just using pip freeze without the redirection. It should be exactly the same. Now if we ever wish to reinstall our Python libraries then we can just use the requirements.txt file as the input to pip and it will run through the list and install everything, e.g.

```
$ pip3 install -r requirements.txt
```

Now we can get to the meet of Pip, which is installing new Python libraries. We do this using the install argument of Pip, e.g.

```
$ pip3 install flask
```

Obviously we already have flask installed on the dev server but if we didn't then we could use this simple command to just install it.

---

<sup>1</sup>There are some opportunities for further exploration here. The '*>*' symbol is a Bash shell redirection operator which means that it causes the output of bash, which is usually printed on screen to be redirected to another location, in this case into a text file.

# Appendix D

## Python Primer

The goal of the following sections is to give the shortest possible introduction to the most important features of the Python language that you'll need to become familiar with. Obviously Python can do much more, but this is a good starting place. All the examples are using the Python REPL (which is why the prompt starts with '>>>'). If you want to run this code in a script then you might have to make minor adjustments in places.

### D.1 Variables & Data Types

#### D.1.1 Variable Assignment

```
>>> x=5
>>> x
5
```

#### D.1.2 Performing Calculations With Variables

Respectively, addition, subtraction, multiplication, exponentiation, remaindered (modulo arithmetic), and division.

```
>>> x+2
7
>>> x-2
5
>>> x*2
10
>>> x**2
25
>>> x%2
1
>>> x/float(2)
2.5
```

#### D.1.3 Data Types & Type Conversion

Most of the time Python's strategy for handling different types of data is known as "Duck Typing", i.e. if it walks like a duck and sounds like a duck, then it probably is a



duck so treat it as a duck. most of the time this works fine, and Python exceptions are fast and efficient so the general rule is to try to use a variable the way you want to use it, catch an exception, and do something different if necessary. Occasionally we need to treat data in particular ways, for example, *coercing* a variable to act as a string.

```
>>> x=5
>>> x
5
>>> str(x)
'5'
>>> int(x)
5
>>> float(x)
5.0
>>> bool(x)
True
```

Notice that although we started with our variable, `x`, being assigned a number value (an integer to be precise), we can wrap our variable in various functions to cause it to be treated as something other than an integer. Notice that each is subtly different, for example, the `int` is displayed as the number value and nothing more, but the string wraps the value in quotes.

Let's look at a quick example of treating a variable of one types as though it is a different type. If we want to output the value of a variable with an explanatory message such as when we want to say “the value of `x` is ” then add the actual value – then we need to ensure that value so that it is treated as a string so it can added to the end of the message.

```
>>> x=5
>>> y="num is "
>>> y+x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
>>> y+str(x)
'num is 5'
```

Notice that when we try to add the integer to our string we get an error, a traceback. However, when we do the same but explicitly say that our integer, `x` should be handled as a string, we get the desired behaviour.

## D.2 Help

We can get help using the builtin help function, e.g.

```
>>> help()
```

You can pass items into the `help` function such as variables and function names to get relevant information about them, e.g.

```
>>> help(str)
```

Should give you information about the string class.

## D.3 Strings

Speaking of strings. If you want a string, then just wrap it in matching quotes, either single quotes, `'`, or double quotes, `"`.

```
>>> my_first_string = 'hello napier'
>>> my_first_string
'hello napier'
```

### D.3.1 String Operations

Once we have some strings we can do things with them. We can cause them to be handled multiple times, we concatenate them with other strings, and we can check what the strings contain:

```
>>> my_first_string*2
'hello napierhello napier'
>>> my_first_string + " and goodbye"
'hello napier and goodbye'
>>> 'n' in my_first_string
True
```

We can also slice up our strings to work only with particular parts, for example, accessing only the characters at a give, zero-indexed, position, or the sub-string between two positions, or the *head* of the string, from the beginning up to a given index, or the *tail* of the string from a give index to the end:

```
>>> my_first_string[3]
'l'
>>> my_first_string[0]
'h'
>>> my_first_string[4:7]
'o n'
>>> my_first_string[:9]
'hello nap'
>>> my_first_string[5:]
' napier'
```

We can perform many more operations on strings, but this is a good start...

## D.3.2 String Methods

Python strings are objects, and objects generally have methods that you can use to manipulate them. For example, switching the case of the characters that make up the string, either making everything lowercase or uppercase, counting how many instances there are of a given element, replacing particular elements with alternatives, and stripping whitespace from the ends of the string (really useful when consuming strings from other places).

```
>>> my_first_string.upper()
'HELLO NAPIER'
>>> my_first_string.lower()
'hello napier'
>>> my_first_string.count('n')
1
>>> my_first_string.count('e')
2
>>> my_first_string.replace('e', 'x')
'hxllo napixr'
>>> my_first_string.strip()
'hello napier'
```

## D.4 Lists

Lists are an incredibly important aspect of Python programming<sup>1</sup>.

### D.4.1 List Elements

We use the square brackets, '[' and ']' to enclose a list and we can store anything in a list, even mixtures of different types of data.

```
>>> []
[]
>>> [1,2,3]
[1, 2, 3]
>>> my_first_list = [1,2,3]
>>> my_first_list
[1, 2, 3]
>>> l1=[2, 2.3, 'string']
```

We can construct lists in many ways, e.g. from variables, by directly including values, or from other lists:

```
>>> a='this'
>>> b='is'
>>> my_list=[a,b,'my','list']
```

---

<sup>1</sup>To be honest, as you continue exploring various programming languages and methods for working with data, you'll probably discover that "list processing" is a fundamental method for working with data. Lots of data that we need to work with is presented as sequences of similar objects, events, or datapoints. So having tools to help us to process each element of any arbitrary list is a really powerful tool.

```
>>> my_list
['this', 'is', 'my', 'list']
>>> list_of_lists = [['a','list'], ['another','list']]
>>> list_of_lists
[['a', 'list'], ['another', 'list']]
```

We can then access elements of the list in various ways. Compare this to how we performed operations on strings earlier and consider how strings are really just lists of characters. In the following we access the members of the list using the zero-indexed position of the member. Notice that using the minus sign accesses the lists from the tail instead of the head. We can also slice the list using the ‘.’ operator to access various sub-lists

```
>>> my_list[1]
'is'
>>> my_list[-1]
'list'
>>> my_list[-2]
'my'
>>> my_list[-0]
'this'
>>> my_list[1:3]
['is', 'my']
>>> my_list[1:]
['is', 'my', 'list']
>>> my_list[:3]
['this', 'is', 'my']
```

Return to our earlier “list of lists”, we can also access these *nested lists* using a similar notation:

```
>>> list_of_lists
[['a', 'list'], ['another', 'list']]
>>> list_of_lists[0]
['a', 'list']
>>> list_of_lists[1]
['another', 'list']
>>> list_of_lists[1][0]
'another'
```

## D.4.2 List Operations

We can perform operations on lists, for example, concatenating or joining them together using the ‘+’ operator, multiplying lists using the ‘\*’ operator.

```
>>> my_list
['this', 'is', 'my', 'list']
>>> my_list + my_list
['this', 'is', 'my', 'list', 'this', 'is', 'my', 'list']
>>> my_list * 2
```

```
['this', 'is', 'my', 'list', 'this', 'is', 'my', 'list']
```

### D.4.3 List Methods

We can also use the methods associated with the list class to interact with a list, e.g. get the index of a list element, count

```
>>> my_list
['this', 'is', 'my', 'list']
>>> my_list.index('is')
1
>>> my_list.count('is')
1
>>> my_list.append("which")
>>> my_list.append("is")
>>> my_list.append("great")
>>> my_list.count('is')
2
>>> my_list
['this', 'is', 'my', 'list', 'which', 'is', 'great']
>>> my_list.sort()
>>> my_list
['great', 'is', 'is', 'list', 'my', 'this', 'which']
>>> my_list.remove('is')
>>> my_list
['great', 'is', 'list', 'my', 'this', 'which']
>>> my_list.pop()
'which'
>>> my_list.pop()
'this'
>>> my_list.pop(-1)
'my'
```

## D.5 JSON

I don't go into much detail in the workbook on this topic because it is a data manipulation thing rather than a webby thing, but perhaps an example might help (this was all done in the python command line REPL rather than a script but converting to a python script or using in your Flask app should be trivial):

We can create a Python dictionary like this:

```
>>> d = {'firstname': 'simon', 'lastname': 'wells'}
>>> print d
{'lastname': 'wells', 'firstname': 'simon'}
```

We can import the JSON library and use it to create a JSON string representation of a Python dict as follows:

```
import json
>>> s = json.dumps(d)
>>> print s
{"lastname": "wells", "firstname": "simon" }
```

Notice that the dict and the string appear very similar in terms of output when they are printed? Internally they aren't the same, one is just a string, but the other, the dict, can be accessed by key, e.g.

```
>>> print d['lastname']
'wells'
```

whereas doing the same with the string gets us an error, e.g.

```
>>> print s[lastname]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'lastname' is not defined
```

We can do the reverse, i.e. get a dict back from a JSON encoded string like follows:

```
>>> d2 = json.loads(s)
```

Now we can compare the original dict, d, with the new one, d2:

```
>>> print d2
>>> {u'lastname': u'wells', u'firstname': u'simon'}
>>> print d
{'lastname': 'wells', 'firstname': 'simon' }
```

The main difference is the u which indicates that the strings in the dict are explicitly unicode encoded. We can also access our elements of the two dicts to compare them, e.g.

```
>>> print d['lastname']
'wells'
>>> print d2['lastname']
u wells'
```

So that is converting between Python dictionaries and JSON encoded strings. Now we want to serialise our string to a file then read them back in. To write our string s to the file testfile.json we can do this:

```
>>> with open('testfile.json', 'w') as outfile:
...     outfile.write(s)
```

Now open testfile.json in a text editor, like vim, or just use cat to show the contents and check it is the same as the dict/string that we had earlier. I made a small edit to testfile.json in the text editor to add a new key:value pair, e.g. middlename:none so that the information loaded back in differs from the original string written out. Now we just need to read the json file back into a dict within our program so that we can manipulate it, e.g.

```
>>> with open('testfile.json') as infile:
...     new_dict = json.load(infile)
...     infile.close()
...     print new_dict
...
{u'middlename': None, u'lastname': u'wells', u'firstname': u'
  s i m o n  }
```

The only real challenge now is to design the right dictionary structure to capture the information you want to store, i.e. data about your collection.

NB. <https://jsonlint.com/> is really useful for testing that a bit of JSON is valid and <https://docs.python.org/2/library/json.html> documents the Python JSON library

# Appendix E

## Git

This section will eventually contain a quickstart on Git to use as an aide memoire.

### E.1 Bare Repositories

**NOTE:** We used to use bare repositories during the submission of the coursework. However we no longer do that and so this section is only here for historical interest (or in case you find that you have a need to use a bare repository, for example, if you want to use your own server for collaborative code development instead of using a service like GitHub).

We need to create a Git repository that can be used as a remote repository, somewhere where you can store your own code, but from which others, like your markers, can retrieve your work. A bare repository is the Git terminology for such a remote Git repository.

There are actually two basic kinds of Git repository, *working* repositories and *bare* repositories. The kinds of repo that you have most likely worked with so far are working repositories. These are repositories in which you can add, edit, and delete files. Most files that you clone from other locations, like GitHub or BitBuckets are working repositories. Bare repositories are not designed for working in, they are missing the *working tree*, the place where you would add, edit, and delete files. It is easy to convert between a bare and working repository so if you have one, then you can easily create the other type from it, i.e. cloning from a bare repository will create a working repository but cloning with the `-bare` argument from a working repo will give you a bare repo.

So what is the purpose of the bare repository? Well, a bare repository is meant to serve as an authoritative focal point for collaborative development. This means a place from which other developers can clone, fetch, or push changes. The idea is to minimise the number of problems that can occur when working collaboratively, and the easiest way to do this is to ensure that copies of files associated with “working” repositories are not stored in the exact same place as the repository around which collaborative development is focussed.

This basically means that, for submission of our assignment, to make it available to other users of your VM such as your markers, we need to create a repo that can be shared between multiple people (assuming they have access to the machine that the repo



is stored on, otherwise some additional steps might be required). Note that you don't do work directly in a bare repo, you must clone it to a working location then push any changes that you've committed back to the bare repo.

Create a folder in your home directory and name it something useful like test-bare-repo, then change directory into your newly created location e.g.:

```
$ mkdir test-bare-repo
$ cd test-bare-repo
```

At the moment this is just a regular folder on the filesystem but Git can add files to turn it into a Git repository. To turn this into a git repository we need to do the following:

```
$ git init --bare .
```

In order to use this repository you will need to clone it to a working location on your VM (or on another machine if you are developing in another place and only deploying to your VM).

**Note** If you *already* have a Git repository set up that you want to use then you can make a bare clone of that repository using a variation on the above command, e.g.

```
$ git clone --bare some-repo-name bare-repo-name
```

which should give you a message like:

```
Cloning into bare repository 'working-repo.git'...
done.
```

As we noted earlier, we can't work directly on a bare repository, we have to clone it to a different location to work on it's contents. Once we've made changes to our working repo, then we can push the changes back to our bare repo. The following sections are a small indicative demonstration of using our bare repo. You might have to adjust paths and filenames to match your actual setup.

Starting in the bare repo folder run git log, e.g

```
$ git log
```

You should get output similar to the following:

```
fatal: your current branch 'master' does not have any commits yet
```

This is just letting us know that our bare repo is empty and hasn't yet got any commits. Note that you can't make changes directly to a bare repo. Instead you should clone it to another location, a working location, then push any changes to your working copy back to the bare repo. Let's do that.

First we need to make a working copy of our bare repo. Change directory to another location in your account then clone your bare repo to create a working repo:

```
$ git clone test-bare-repo working-repo
```

This will clone our initial bare repo into a working repo called "working-repo".

You can now move into your working repo and treat it like a regular Git repo, e.g. adding files, committing them, and then pushing them back to the bare repo. Let's see an example of that. First move into our couserwork working repo and create a new file:

```
$ cd working-repo
$ touch hello.txt
```

Now edit the file, then add it your repo:

```
$ git add hello.txt
$ git commit -m "Our first file"
```

If you run git log now you should see a message like this:

```
commit fffd9f14062f39a4abc7360d0b342e7979319277 (HEAD ->
    master, origin/master)
Author: Simon Wells <siwells@gmail.com>
Date: Thu Nov 5 07:33:18 2019 +0000

    Add first file
```

Now we have made changes to our working repo but we want to push those changes back to our bare repo so that those changes are available to other members of the team. We can do that like this:

```
$ git push
```

Which should yield a message similar to this (your paths, file, and repo names will likely differ):

```
Counting objects: 3, done.
Writing objects: 100% (3/3), 226 bytes | 226.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To /Users/simon/Desktop/_tmp/test-bare-repo
 * [new branch]      master -> master
```

You can now change directory to your bare repository and run `git log` again to verify that your updates to the repo are reflected accurately in the bare repository.

Anyone who has read access to the folder on your VM that contains your bare repo can clone it. This includes the markers for this module. You should now create a bare repo in the location specified in the assignment descriptor, e.g. at the following complete pathname:

```
/home/<matriculation-number>/submission/project.git
```

NB. Recall that your home directory is `/home/<matriculation-number>` so you just need a new sub-directory called `submission` then a sub-directory of that called `project.git` which is then initialised as a bare Git repo.

You should also verify that the bare repo can be cloned to another machine using the command specified in the assignment descriptor, e.g.

```
$ git clone matriculation-number@webtech-NUMBER.napier.ac.uk  
:~/submission/project.git
```

NB. You will need to add in your own matriculation number and VM number to complete this command correctly. Try cloning your repository from another machine to verify that it works. You can use a tool like `git-bash` if you are on a lab machine. It's a good idea to do this way in advance of the hand-in. If you have trouble, ask a lab demonstrator for assistance.

Two things to note. Firstly, you can set up multiple remote repositories, each of which can act as a backup of your repo and the files that it contains. You can even add GitHub or BitBucket repositories as remotes which can be useful. Secondly, This approach, using a Git remote, is nothin special, it's a standard way to set up a Git repo so that it can be shared between multiple developers, so if you need to do a team project, but you don't want to risk the security or privacy of your code by hosting on a cloud service, then this is one way to solve the problem.

# Appendix F

## Deployment

The final step in our advanced web technologies journey is to deploy the apps that we've been building all trimester. Whilst the way that we've currently been deploying our apps during development, using the flask debug server, is perfectly fine for development and debugging, there are a few problems:

1. Our app isn't available on port 80 - so it is not located in the place that our browser would look at by default. Our users must know to choose a different port and must specifically indicate that in the browser.
2. Our app stops running when we log out. At the moment we run our app from the command line as a user-level program. What we really want is for the operating system to manage our app after we log out from the VM. A corollary to this is that we also want our app to be restarted if it ever dies, and for everything to start working again automatically if the server is rebooted (for example when IS applies monthly security updates).
3. The debug server isn't meant for anything more than testing that your code works. It is not built for stability, scalability, or security.

We can solve all of these problems by using appropriate web server software to host our app, and making sure that that server software manages the service lifecycle of our app and makes it available at an appropriate public address. Basically we need a web-app server that understands the WSGI protocol that Flask uses to communicate the functionality of a Python web-app to the wider world of web software. There are a variety of choices for this,  $\mu$ WSGI is popular, robust, and very tunable so it can be optimised for scalability and eke'ing out every ounce of performance from your hardware, but this leads to it being tricky to set up and deploy. Instead we'll use Waitress which is quicker and simpler to set up to cover the middle ground of serving up a web app when we don't yet know the performance characteristics that we want to optimise for. Note that WSGI app-servers are usually not exposed directly to the public, but run privately within a server, inaccessible to the outside world. A *reverse proxy* is then used to relay communications from the internal, private web-app, to the outside world. This is done for a number of reasons, because traditional static web servers, which are usually chosen to act as reverse-proxies, are well-tested and reliable, and only serve up static resources, they represent less of a security risk if something goes wrong. Because WSGI apps are generally dynamic, and executing perhaps less well tested code, there is more of an attack

surface for them. By hiding as much of the WSGI app in private and only exposing the minimum necessary, risks are reduced but not eliminated. We also need a web server that can work as the public facing side of our deployment, a task that is usually given to a traditional, static web-server like NGinX or Caddy. This front-end, public-facing server will act as our reverse-proxy that we just mentioned. The role of the reverse proxy is to take any incoming requests and relay them to our WSGI/Flask app, then take the responses from the WSGI/Flask app and relay them back to the original caller who made the request. We will use Caddy as our reverse-proxy/front-end, static web server. This is for similar reasons to our choice of Waitress for the WSGI app-server. Caddy is simpler to set up than NGinX and covers a happy middle ground for general purposes until you know what you need to optimise for. Note that NGinX is excellent, a very scalable, fast, robust, and reliable tool, but it can be finicky to set up.

The following steps will take you through the process of setting up and deploying a small demonstration WSGI application. The WSGI app will be served by internally and privately by Waitress, and Caddy will be the externally facing static web-server that reverse proxies requests and responses between our WSGI app and the outside world. I've created a set of scripts to automate most of the setup process and a demonstration web-app as a known good starting app for the servers to publish.

It is easiest at this stage to download the zip file from github that contains our deployment scripts and demo app. We want to download it to the root folder (i.e. /home/MATRICULATION-NUMBER/ ) of our VM account. Remember that you can always get directly to the root folder location just by typing *cd* and hitting return. Don't move the zip files elsewhere as the scripts expect things to be in specific places in order for everything to run properly. So let's get the zip archive first:

```
$ curl -o deployment.zip https://siwells.github.io/deployment
/deployment.zip
```

You might need to install the zip tool, which also installs unzip, so that we can open the zip archive. We only need to do this once on a server and then it is installed and available forever, unless you uninstall it, so if you've already installed zip, then skip this step:

```
$ sudo apt install zip
```

Now just unzip the archive. Remember to do this in the root of your account.

```
$ unzip deployment.zip
```

This will have extracted two folders, once called "assignment" and one called "scripts". First we're going to deal with the assignment folder. It contains a demonstration Flask app that does just one thing. Every time it is called it will generate and return a unique ID code. This is useful when testing as every refresh will give you a new value so you can see if things are working if you're unsure.

Setting up this demo app is straightforward if you've worked through the rest of this workbook, or at least the first few chapters, so it should all look familiar. There are a number of commands that we'll execute to change directory to the assignment folder, create a new virtualenv<sup>1</sup> called "env", start the new virtualenv, then finally install the requirements for the Flask app using pip and a requirements file. The requirements are just Flask and Waitress. Note that Waitress is just another Python application that we install into our virtualenv.

```
$ cd assignment/  
$ virtualenv env  
$ source env/bin/activate  
$ pip install -r requirements.txt
```

We can now start our app to test that it works and also so that you know what it should look like when it runs:

```
$ flask --app assignment run --host=0.0.0.0 --port 5000
```

Two things to note about this demo Flask app are firstly that it is called `assignment.py` and that the Flask app object within it is called "app". This is important because we need to pass this information, the name of the package containing the app object and the name of the app object into Waitress later. This will be taken care of by our scripts but if you want to deploy a differently named project later on then it is useful to know how Waitress invokes our Flask app.

At this point though we haven't actually done anything new to get this new app served publicly. That'll be our next step, so, having setup the app that we're going to serve using Waitress and Caddy. We can now go about that process of actually getting Caddy and Waitress working together. Change to the scripts folder that you extracted from the zip archive earlier, i.e.

```
$ cd ~/scripts/
```

In the scripts folder are five files. One of these is a configuration file for Caddy, and the other four are scripts to install and setup first Caddy, then Waitress. We are going to execute these in sequence. Unless something fails you should only need to do this once as they install software and make permanent system wide changes. These scripts wrap up a bunch of commands that you *could* execute yourself and that, you might, later want to adjust, for example, to point waitress towards a different flask app to run. Note that I've only set this up to run a single Flask app, but you could, in theory, have Caddy point to multiple different Flask apps, as well as also doing static hosting, but I've tried to keep things as simple as possible.

This first script essentially does a system wide install of Caddy. As Caddy is not in the default Apt repositories used by Ubuntu to install software, we need to retrieve the correct cryptographic keys and additional repositories. This is all done automatically for us by the *install-caddy.sh* script.

---

<sup>1</sup>I assumed that virtualenv is installed on your VM, otherwise also run *sudo apt install virtualenv*

```
$ ./install-caddy.sh
```

At this stage we can check whether caddy is installed and working by visiting our VM in a browser. Just enter the full VM name (e.g. `http://webtech-2324-XX.napier.ac.uk/`), or the IP address if you know it, into your browser's address bar and you should see this:

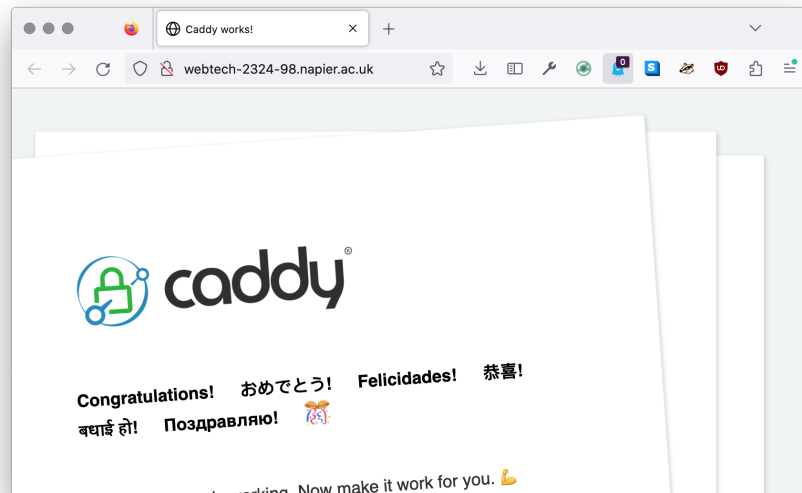


Figure F.1: The default Caddy welcome web page.

The second script we need to run will transfer a new Caddy configuration file, called a Caddyfile, to the correct place. This new Caddyfile will do exactly one thing. It will stop the default static file web serving that we just saw and instead will redirect every request to localhost port 5000. Basically it is this configuration that causes Caddy to act as a reverse proxy to expose our Flask app. However because we haven't yet get our Flask app running permanently, you'll just get a blank screen for the moment<sup>2</sup>.

```
$ ./deploy-caddyfile.sh
```

At this point Caddy is set up and will now, permanently forward all requests to localhost 5000 even if you log out, and it will start doing this again automatically if you reboot the machine. So now we need to move on to setting up waitress. This first script generates a service file for the software, called *systemd* that Ubuntu uses systemwide to control all of the services that the operating system performs. This service file tells systemd to invoke the specific instance of waitress that is in our virtualenv and to use that to load our assignment app. These lines are where that magic happens:

```
1 WorkingDirectory=/home/MATRICATION-NUMBER
2 ExecStart=/home/MATRICATION-NUMBER/env/bin/waitress-serve --listen
  =127.0.0.1:5000 assignment:app
```

<sup>2</sup>although if you run your flask app as we did earlier, you should see the output from it

In the first line we tell systemd where to run the flask app from, i.e. your home directory. The second line then invokes the version of waitress in the specified virtualenv, tells it to listen on localhost port 5000 and specifies that the specific app object to invoke is called app and is in the assignment package (i.e. assignment.py). Let's just go ahead and generate that service file now (note that there are other directives to systemd in the full service file but the two above were the ones that tell systemd where to find our app and how to run it):

```
$ ./generate-waitress-service.sh
```

Finally, we need to move the service file we just created in the previous script, to the place that systemd looks at for service files, and then we tell systemd to use this service, to start the service after a reboot, and also to start it now:

```
$ ./deploy-waitress-service.sh
```

At this point we should be able visit our VM, again by IP address or domain name, and see our site without adding in any additional port numbers. Additionally, the site should, barring any coding errors on your part, stay up permanently. Note that I've called the demo flask app "assignment.py" for a good reason. This should make it easier to deploy your own web-app developed during the assignment. In theory you could just put your own assignment code into the assignment folder and so long as the main python file is called assignment.py, it should just work<sup>3</sup>

Note that this is for *deployment*, i.e. to have your web app up and running permanently, and isn't meant to be used for development. So if you are still developing your app, then you should continue to use the Flask debug server until all features are implemented and ready to be made public.

Finally, if you got this far, and everything is running successfully, then **congratulations**, you just created and deployed a dynamic website to the public Web.

---

<sup>3</sup>experiences may vary....