Edinburgh Napier University

SET09103 - Advanced Web Technologies

Topic 04 - Frontends

Dr Simon Wells

1 Overview

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

1. HTML,
2. Static Files,
3. Uploads,
4. Templates (Jinja 2)

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

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

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

The practicals are built around the following basic structure:

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

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

2 Activities: Exercises

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

2.1 Getting Started

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

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

So far we've ...

2.2 Serving HTML Files

I don't know about you, but keeping all of the HTML content for our sites inside strings within our Python methods seems like a terrible way to do things. That's because it is. The best place for HTML content is in an HTML file. It turns out that, with a little bit of configuration, Flask will serve HTML to our clients just like a traditional static web-server[2].

In a traditional web-server, the HTML is stored in files on disk and the server is optimised to respond to requests for files really quickly and reliably. These files don't change as the pages are not dynamically generated. Once we consider static files, then any content can be considered for distribution by the Web server, for eaxmple, image files, audion files, video files, and arbitrary data files, but right now we'll deal solely with static HTML. This is mainkly because HTML files are treated a little differently to other static files. So we'll deal with HTML as a static file now, then consider other kinds of static file, and later we'll return to our HTML files and consider them as templates for *incomplete* web pages which can contain code which enables them to be dynamically altered during a request-response cycle to generate their final complete HTML page that is returned to the client.

We're now going to build yet another Hello World app, but this time we're going to put the HTML content into an actual html file[3] outside of our web app and then get Flask to return that file on request.

This version of hello world is almost identical to the previous one, however, to make things easier, we've moved the HTML content for the page out of the Python script, and into it's own individual HTML file. If you have built more elaborate sites in any of the exercises or challenges, then you might have one html file for each page. Note that later when we look at templates we will learn to use them to create a commmon design so that multiple pages can use the same basic template, but for this specific exercise, one page equals one template.

These HTML files are stored in their own sub-folder named 'templates'. The templates folder is the default location[4] in which Flask looks for HTML templates. Flask will serve up any matching HTML file in the template folder when the render_template() function is called with the name of the template as an argument. Note that a template is an HTML file that possibly contains additional JINJA2 code directives to dynamically alter the HTML content. However, we don't have any Jinja2 directives in our HTML files at this point, so our templates are the most straightforward that we can create, raw and valid HTML, which will be returned unaltered to our calling client.

Organise your project as follows. In your project folder for this exercise create a hello-static.py script and also a sub-folder called "templates". Into the templates folder you will save your HTML files that represent the pages of your site.

The 'hello-static.py' file listing is as follows:

```
1 from flask import Flask, render_template
2 app = Flask(__name__)
3
4 @app.route("/")
5 def hello():
6     return render_template('index.html')
```

This is a simple Flask app, very similar to the previous Hello World example, but we've extracted the HTML from within the Python file and placed it in it's own separate html file. I've actuall gone a little further than that in fact and supplied all the missing tags so that the html file is valid. Note that the Flask app won't work properly untless you've also created your index.html file. Let's take a look at the contents of the 'index.html' page:

[2] just remember that the Flask development server is not as well optimised as web servers that have been designed for static files only. Nginx, Caddy, and even Apache, are better static Web servers for general purposes

[3] basically just a file whose named ends in .html and whose content is HTML code.

[4] although this can be overridden

```
<!DOCTYPE html>
<html>
    <head></head>
    <body>
        <h1>Hello World!!!</h1>
    </body>
</html>
```

To send a specific HTML file back to the caller we use the render_template() function passing in the name of the template to send. When we create a route, we can associate that with different templates, so that each corresponds to a different page of the site. When a route is called then a different template, and hence, a different Web page, will be returned.

This basic approach can be used to get Flask to serve an arbitrary number of pages, on a variety of routes, where each page corresponds to a named template in the *templates* directory. We'll build on this again quite soon to see how we can add additional code to the html files in the templates directory so that we can dynamically alter the content. For the moment though, think of this as static hosting of HTML files using Flask.

2.3 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 specical folder whether 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 parameteres 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 2.5.

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 them place your static files, your CSS, JS, image files, &c., into this folder and Flask will automatically look there for them when an HTML files requests them[5][6]. You can, and should organise files within your static folder though. Create sub-folders to so that your static files are well organised. This is particularly important as the size of your site, 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.

[5] 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.

[6] Remember that an HTML file requests other files, like CSS using the <link> tag and JS using the <script> tag

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 dyanamic 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[7].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. Download the following image file: `siwells.github.io/assets/images/vmask.jpg` then move it into our static sub-folder.

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 = '<img src="'
11   url = url_for('static', filename='vmask.jpg')
12   end = '">'
13   return start+url+end, 200
```

Notice how we have used a standard HTML image (img) tag, filled in the src attibute using the output from url_for, then concatenated the three strings together to form the string that is returned by the static_example_img function. I wonder what other HTML tags could be used to return information from a URL? Perhaps we could build entire web-pages this way, by just concatenating various sections of HTML to make an entire page....[8].

[7]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 techniquess for deploying Flask web-apps that are generally applicable to most web sites.

[8]You can do this, and you can return *any* HTML tags this way and build up quite complex pages. However in chapter 2.5 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.

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

Figure 1: 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.

2.4 Uploading Files

Let's now take a short diversion and consider a topic that intersects both with the current one regarding files, and with our previous topic on client requests, the matter of file uploading.

We will occasionally want to enable our users to upload materials to our site. One scenario is if you have implemented user accounts on your site and want to allow every user to submit their own image to use as their personal ident.

We can support file uploading by ensuring that two things occur. Firstly, 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, secondly, we must do something with the transmitted file. Otherwise the file will sit by default either in memory or in temporary storage rather then being saved for later reuse.

In the following I used a PNG image file that I had available and renamed upload.png 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')
```

```
    return "File Uploaded"
  else:
    page='''
    <html>
    <body>
    <form action="" method="post" name="form" enctype="multipart/form-data">
      <input type="file" name="datafile" />
      <input type="submit" name="submit" id="submit"/>
    </form>
    </body>
    </html>
    '''
    return page, 200
```

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 2.3. For example:

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

@app.route("/display/")
def display():
  return '<img src="'+url_for('static', filename='uploads/file.png')+'"/>'

@app.route("/upload/", methods=['POST','GET'])
def account():
  if request.method == 'POST':
    f = request.files['datafile']
    f.save('static/uploads/file.png')
    return "File Uploaded"
  else:
    page='''
    <html>
    <body>
    <form action="" method="post" name="form" enctype="multipart/form-data">
      <input type="file" name="datafile" />
      <input type="submit" name="submit" id="submit"/>
    </form>
    </body>
    </html>
    '''
    return page, 200
```

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.

2.5 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[9] which is already installed on the Linux dev server alongside Python and Python-Flask.

[9] https://palletsprojects.com/p/jinja/

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.

2.5.1 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:

```
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)
```

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

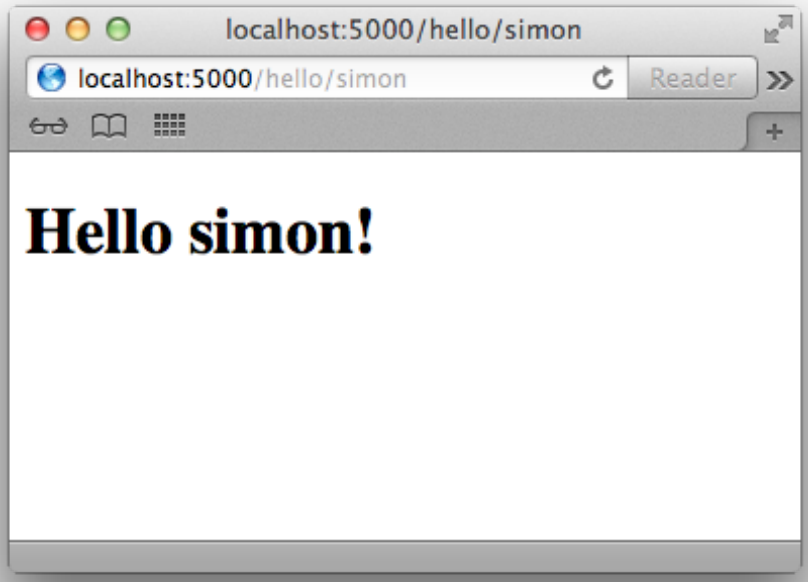

Figure 2: Rendered HTML with a very simple template & a single parameter

2.5.2 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:

```
<!doctype html>
{% if name %}
    <h1>Hello {{ name }}!</h1>
{% else %}
    <h1>Hello from Napier!</h1>
{% endif %}
</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:

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

@app.route('/hello/')
@app.route('/hello/<name>')
def hello(name=None):
    return render_template('conditional.html', name=name)
```

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:

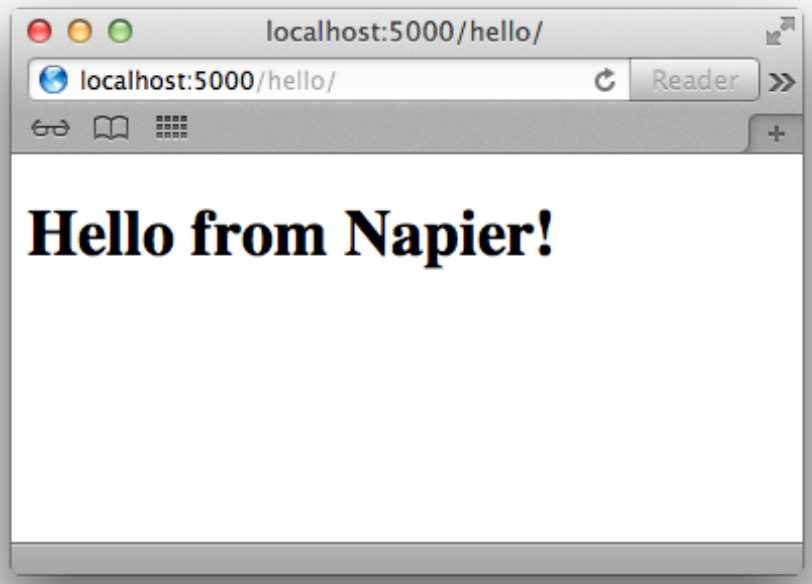

Figure 3: Conditional template rendering without URL arguments

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

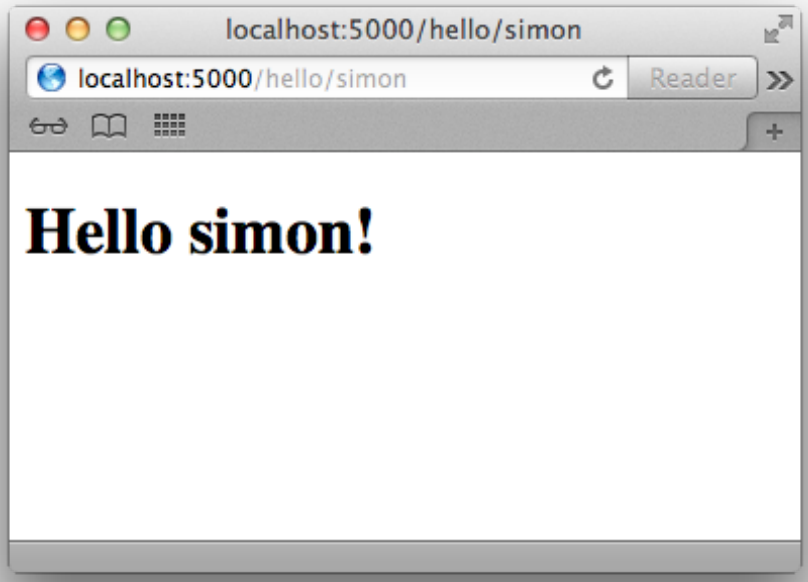

Figure 4: Conditional template rendering with a single URL argument

2.5.3 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:

```
<!doctype html>
<body>
  <ul>
    {% for name in names %}
     <li>{{ name }}</li>
    {% endfor %}
  </ul>
</body>
</html>
```

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

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

@app.route('/users/')
def users():
  names = ['simon', 'thomas', 'lee', 'jamie', 'sylvester']
  return render_template('loops.html', names=names)
```

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.

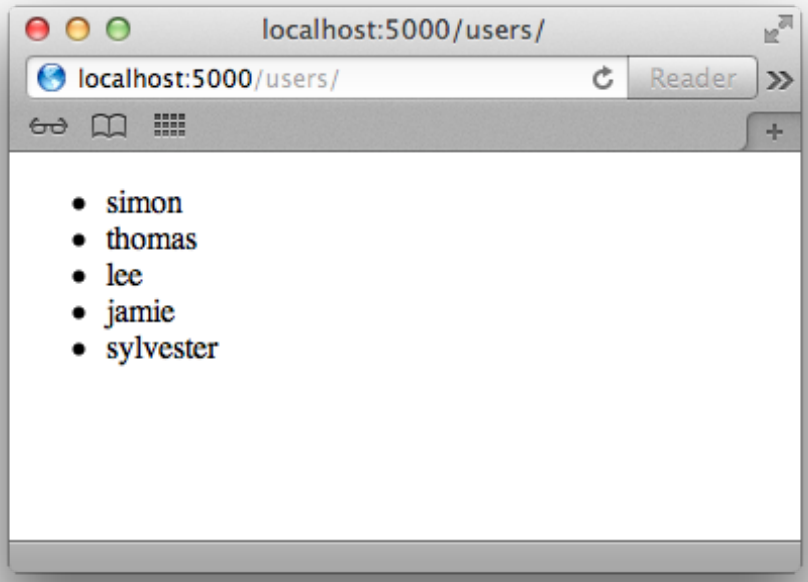

Figure 5: Looping over data in within a template to generate HTML

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.

2.5.4 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:

```
<html>
<head>
    <title>Template Inheritance Example</title>
</head>
<body>
    <h1>Title stored in the base template</h1>
    <h2>With a subtitle</h2>

    {% block content %} {% endblock %}

</body>
</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:

```
{% extends "base.html" %}
{% block content %}
  <p>First example template. It contains some stuff</p>
{% 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 pone.

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

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

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

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

@app.route('/inherits/one/')
def inherits_one():
    return render_template('inherits1.html')

@app.route('/inherits/two/')
def inherits_two():
    return render_template('inherits2.html')
```

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 reptition 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.

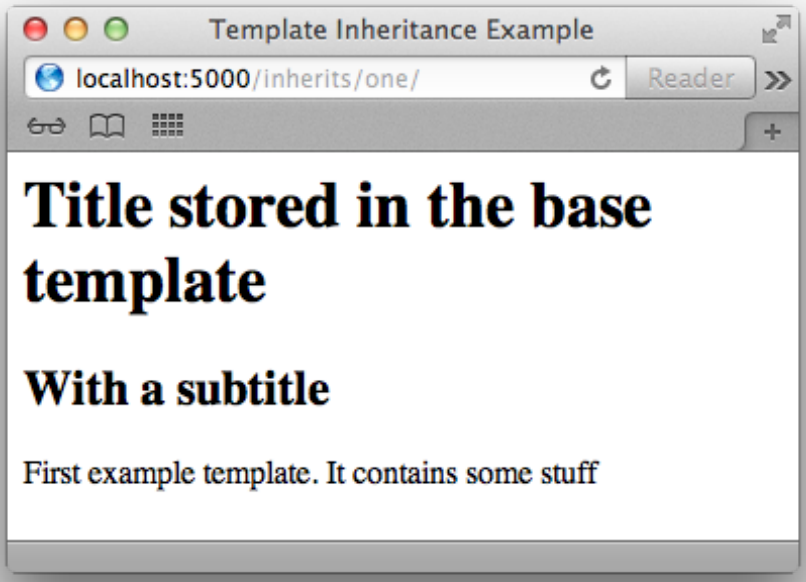

Figure 6: The first page that inherits from our base template

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.

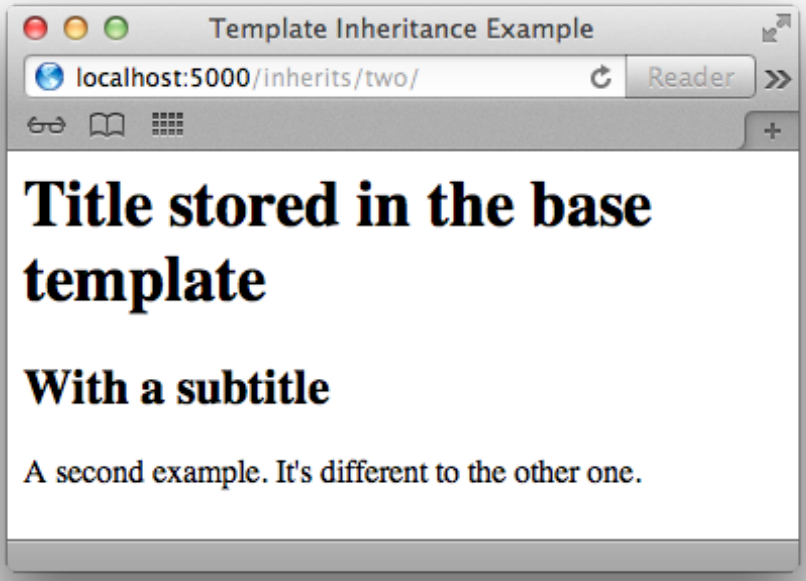

Figure 7: The second page that inherits from our base template

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:

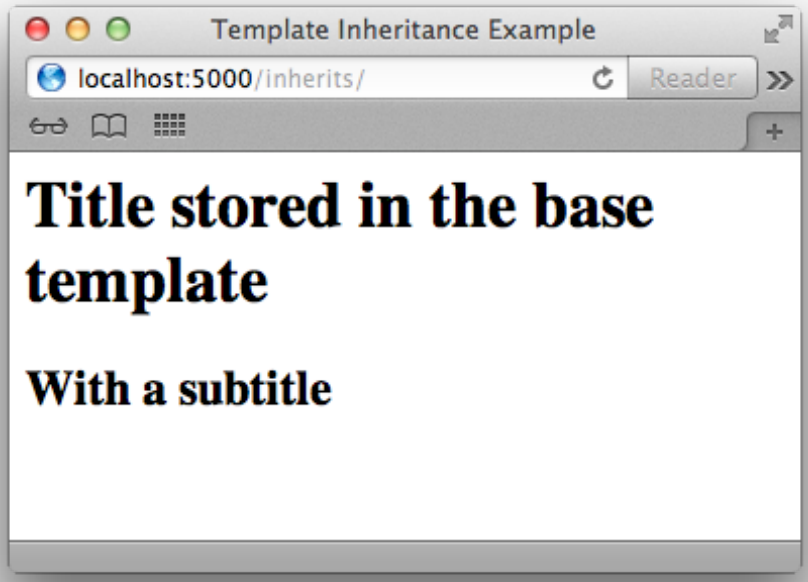

Figure 8: The rendered base template

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.

3 Activities: Challenges

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

1. For any, or all, of the sites you've created so far, where you've returned pages to the client by returning a quoted string, create a new version in which each page is a distinct HTML file stored in the templates folder of your project. The CIA World Fact Book site and the Choose Your Own Adventure site are both good places to start.

2. Elaborate on your site from the last challenge. Add some CSS and perhaps some other static files, like some images, to make the site more visually appealing.

3. Design and implement a simple *Dungeon Crawler Game*. Start simple at first, perhaps an entrance/exit, and a treasure room (the goal), and a small number of intermediate rooms. Each room should have a description. Use a template to represent the main game playing interface, but supplying a different description to each room as you navigate between them. Use links for the navigation between rooms. Use Jinja2 variables in your templates to hold the description of each room and to hold the link destinations so that you can supply the content from Python once each route is called. Consider, and then perhaps implement, anything that might make this game more fun and engaging.

If you're still not confident with creating a new Flask web-app from scratch[10] don't just edit and re-use your existing ones. The setting up aspect is still part of the practise. For each of the exercises, create a new project folder in your set09103 folder. You will need to create a new virtual env each time and install pip. Each project folder should have a unique name but don't call it app or flask as this will cause errors. Doing this will give you practise in setting up new Flask projects

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

so that it becomes straightforward. We'll be doing that a lot over the coming weeks so making it less of a chore is a good idea. One tactic that can help is to create yourself a crib sheet containing the *invariant* commands that you use each time you set up a new Flask app for development. Over a short period of time, and a few repetitions of practise, you **will** get past this.

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

4 Finally...

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

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

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

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

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

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

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