David Wheatley (01633983) **11 May 2020**

Fundamentals of Database Technologies

– Assignment 5 –

1. Introduction

This paper details the steps taken to create a simple web application using a Python Flask Server and deploying to the Heroku hosting platform using the Git version control tool. The application uses the Python SQLAlchemy library to connect to a Postgres database.

This paper is split into 4 sections

- An overview of the tools used
- Building a basic web application
- Deploying to the Heroku platform
- Connecting to a database

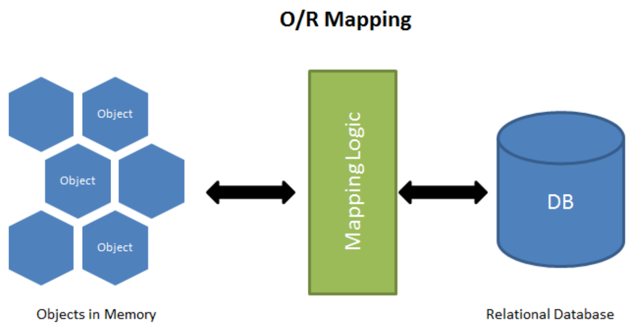
2. An overview of the tools used

This section provides an overview of the tools and the value they add to the process of developing and deploying an application.

2.1 Object-Relational Mapper (ORM)

An Object-Relational Mapper (ORM) provides an interface between a database and an object-oriented programming language. Databases are used extensively to store, share and maintain vast quantities of data across a wide variety of applications.

While a database engine enables applications to directly interrogate, amend and extract data from database tables, an ORM enables the data to be stored in efficient data structures in memory, reducing the database load and enabling a simpler integration into the wider application.



Within an object-oriented programming language, classes define a common set of properties and methods that can be applied to an object. An ORM maps an entity in a data table to an object, enabling a range of queries such as joins, add, amend, and delete to be performed and, where necessary reflect these changes back into the original database. ORM's also manage the interface with different database dialects (e.g. Postgres MySQL, Oracle) via a variety of database engines, simplifying the demands on the developer.

Object-oriented programming is particularly suited to managing large volumes of data due to 3 basic tenants:

- Encapsulation, reducing the risk of data corruption.
- Abstraction, hiding the detailed code, making it easier to maintain and improve.
- Inheritance. Relationships and subclasses can be assigned to objects, allowing common logic to be reused.

Despite the above benefits ORM's may not provide the optimal solution for complex, intensive applications due to the additional processing overhead they provide, which can have a detrimental effect on speed. Furthermore the ability to translate between different database dialects precludes the use of some of the more nuanced features the individual databases provide.

2.2 FLASK server

Flask is a lightweight Python web application framework. It was created by Armin Ronacher in 2010 and is used by a variety of companies including Netflix, Reddit, Airbnb, Lyft, Mozilla, MIT and Uber. The name 'Flask originates from the fact it was developed as a wrapper around two pre-existing Python libraries, Werkzeug and jinja 2.

- Werkzeug is a Web Server Gateway Interface (WSGI) utility library, its principal task is to define how the web server communicates with the application.
- Jjinja2 is its template engine enabling static HTML files to integrate dynamic elements into a web page. The dynamic elements are controlled by taking a tokenized string from the python code and converting it into values that can then be displayed.

In section 4.2 we will see both these libraries mentioned when we push the application to the host environment.

2.3 SQLAlchemy ORM?

SQLAlchemy is another popular Python library used by many major organisations including Yelp!, reddit, DropBox and Survey Monkey. First developed by Michael Bayer in 2006 SQLAlchemy maps data from a database into objects as defined in section 2.1. The origin of the name come from the word 'Alchemy', meaning transmutation of matter, and the SQL database programming language.

In addition to the core ORM functionality SQLAlchemy provides a more complete database management tool kit which includes:

- SQLAlchemy Relationship Patterns – Enabling the application to understand database table keys and associated relationships, permitting classes to be populated from joined tables
- SQLAlchemy Connection Pools - enables the database connections to be abstracted away simplifying the process of accessing the database.
- SQLAlchemy Dialects - Enables the use of a common API to be used to access a multitude of different database engines through common code instructions. In the example application developed below we use psycopg2 which is specific Python database engine for interacting with Postgres.
- SQLAlchemy ORM Cascade – Enables updates to a table which have an impact on other tables to be automatically updated accordingly
- SQLAlchemy Sessions enables the coordinated updating modifications to the underlying database to help eliminate concurrency issues

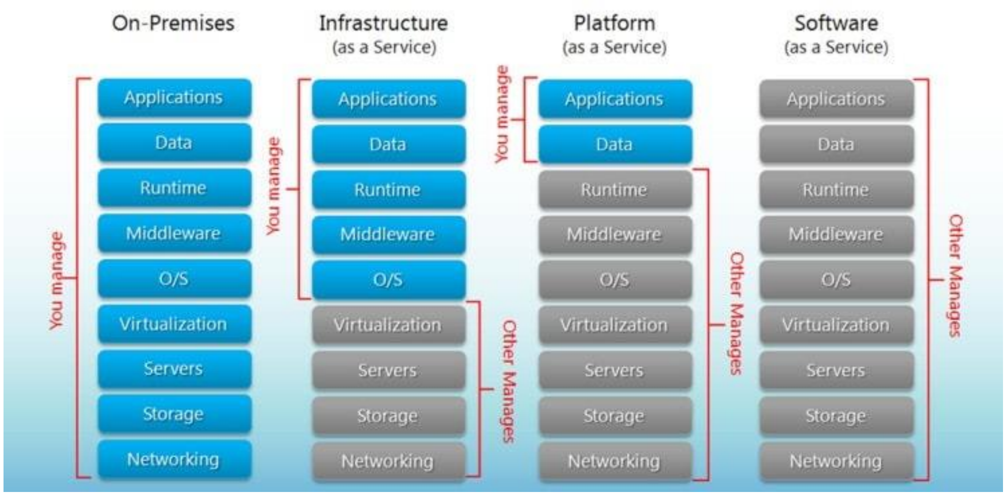
An additional key feature we take advantage of in the application detailed in section 3 is the Flask-SQLAlchemy extension which enables all the SQLAlchemy functionality to be utilised from within a flask server web application.

2.4 Heroku hosting platform

According to their website Heroku "is a container-based cloud Platform as a Service (PaaS)". Founded in 2007 and acquired by Salesforce in 2011 the name is derived from the words "hero" and "haiku" and is a popular hosting solution for many smaller organisations focussed on rapid growth, such as the fintech start-up RocketChart.

The two key terms used to explain what Heroku are PaaS and container-based, so we will now explore each of these in turn.

Over the last 15 years cloud computing has become increasingly prevalent, with organisations seeking to leverage the increased resilience, capability, accessibility and scalability cloud solutions can offer, while also reducing capital expenditure. The spectrum of hosting options available are illustrated in the diagram below.



.

Depending on the desired level of control required the solutions on offer range from IaaS (from providers like Amazon AWS and Microsoft Azure) to PaaS (eg. Heroku and AWS Elastic Beanstalk) to SaaS (e.g. DropBox)

Simplistically IaaS solutions allow the customer to manual configure servers via a dashboard, while PaaS provides ready-made tools you can use but not manage. The advantage of PaaS is that it allows developers to focus on building the applications and spend less time setting up the environment. Applications like Dropbox are examples of a software as a service (SaaS) providing ready built, easy to use solutions via the internet.

A container is an isolated environment into which an application and the associated dependencies are hosted. Creating an isolated environment allows the application to be deployed to different platforms and avoid conflicts with other applications. Each container shares the same host operating system making them much more resource efficient than virtual machines, which require a dedicated operating system. This also mean that representative environments can be created locally simplifying the development process, a feature we take advantage of in section 4.1.

In summary Heroku provides the ability to build, host and maintain bespoke applications in an isolated cloud environment without the overhead of managing the infrastructure on which it sits.

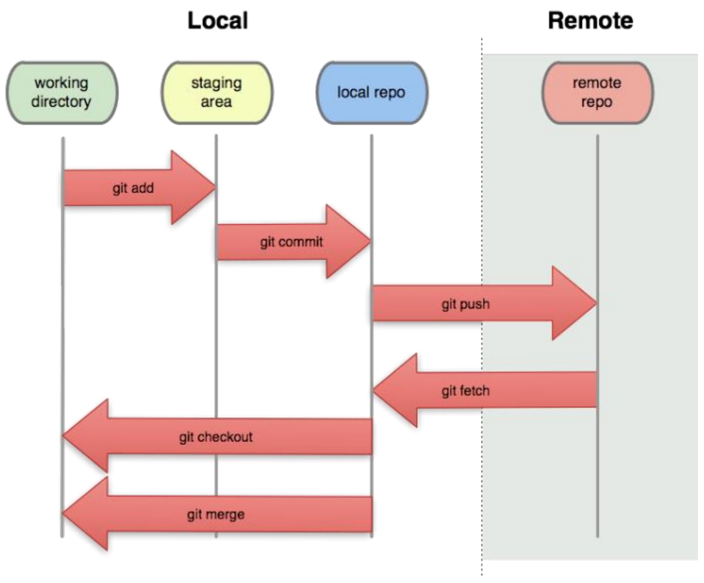
2.5 GIT-SCM?

Git was developed by Linus Torvalds, creator of the Linux operating system kernel in 2005 in response to the needs of a growing Linux development community. Git is widely used by a variety of organisations including Netflix, reddit, and Shopify. The origins of the name "git" are unclear ranging from the pragmatic acronym "global information tracker" to Linus Torvalds claiming he always names his products after himself and in the case of Git because "I'm an egotistical bastard".

Git's key features are:

- Tracking changes, including creation, deletion and modification of files enabling a complete history to be developed and, in the event of a failure, enable roll backs to a previous state to be performed.
- Allowing developers to work concurrently on a common code base by creating and managing separate branches and re-merging them effectively to resolve any potential conflicts.
- Facilitating file sharing through a distributed architecture, allowing developers to push and pull code to and from nodes, thereby reducing the dependency on a central repository.

This is achieved by managing a workflow through 4 distinct phases. These will be used to deploy our application in section 4.3.



A local repository is initialised by copying across the required skeleton files into the working directory using the git init command. Once files have been prepared they can then be set to tracked, using the git add command, which moves them to the staging area.

The commit command then creates a “snapshot” of the tracked files from the staging area into the local repository. During the commit process the developer is forced to add a message detailing the changes made, which is invaluable for understanding the work content during any future bug fixing or in the event a roll back is required.

Up to this point the process have all been performed on a local machine. To make the code available elsewhere the git push command is required to push the code to a remote repository, in the case of our application the Heroku platform.

One of the strengths of Git, that differentiate it from other version control tools is the ability for developers to work collaboratively due to its distributed model. This mean that developers can pull the latest version from any connected remote repository and to push any completed work back, avoiding being dependant on a central repository.

Having introduced the benefits of collaborative development this introduces the risk of code conflicts. To overcome these problems Git, and other VCS, provide the ability to branch and merge the software. The most common strategy used involves maintaining a “trunk” in a releasable state and creating branches to implement new features. After implementing a new feature on a branch and performing the required unit testing the branch can safely merged back into the main trunk and any conflicts resolved.

3 Building a basic web application

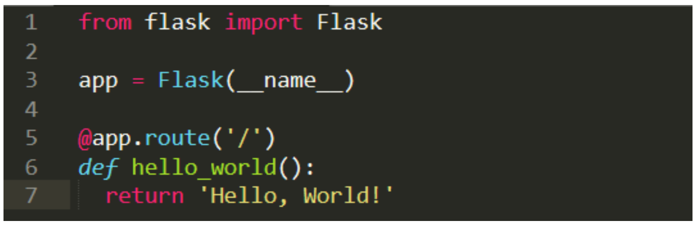
The following sections detail the steps taken to build a simple web application

3.1 Building the basic application

At the command line the flask library is installed using the following command.

> python -m pip install flask

The following simple python file is saved as 'app.py' into a new directory.



```python
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello, World!'
```

3.2 Running the application

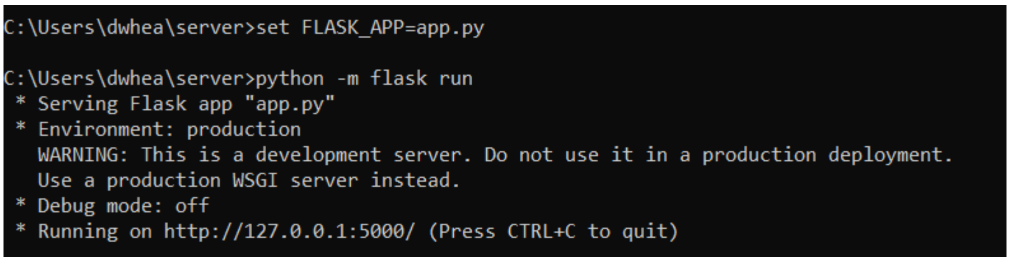
In order to run the above file on a flask server the environment variable FLASK_APP needs to link to our application. This is set using the following command.

> set FLASK_APP=app.py

Now when we run flask our application will be served locally on http://127.0.0.1:5000. This is done be typing the following command.

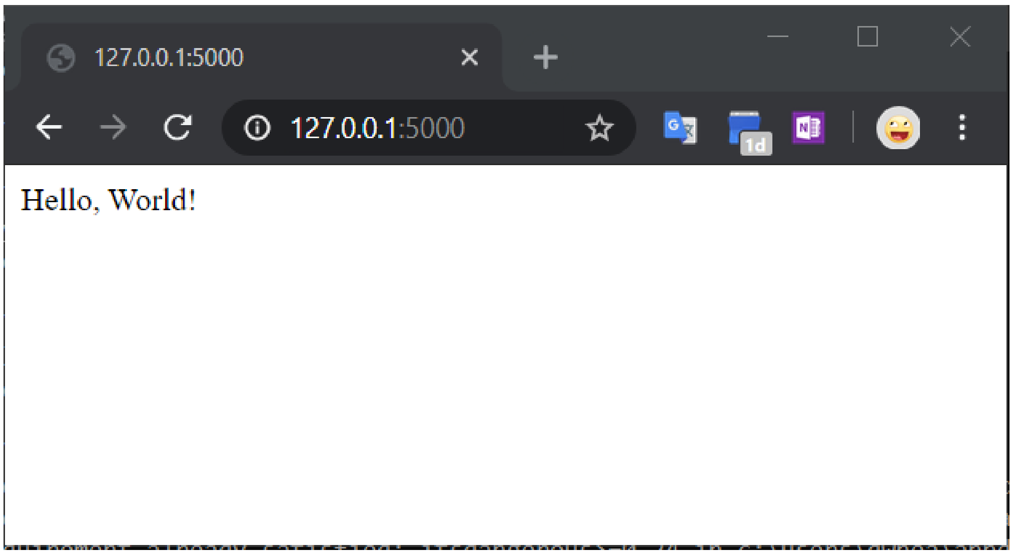
> python -m flask run

After which the following confirmation will be shown



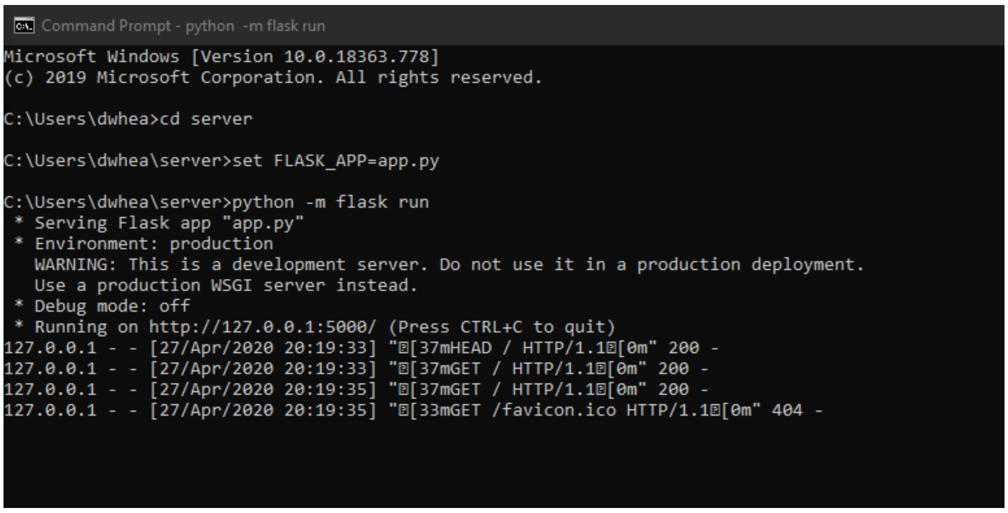
```
C:\Users\dwhea\server>set FLASK_APP=app.py

C:\Users\dwhea\server>python -m flask run
 * Serving Flask app "app.py"
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Typing this URL into a browser runs our application.



Hello, World!

Referring to the command line the following report is produced detailing the connection.

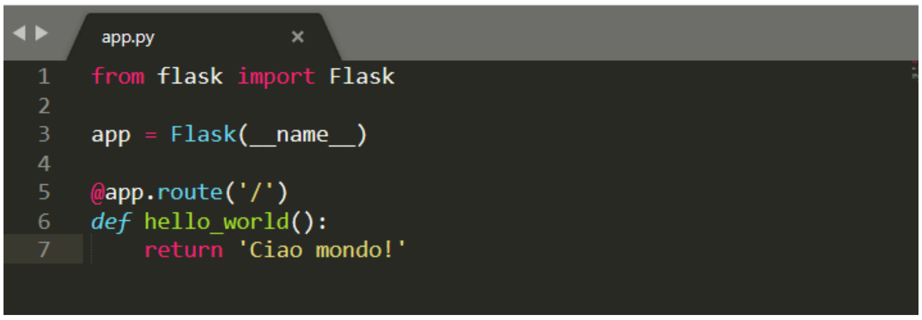


```
Microsoft Windows [Version 10.0.18363.778]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\dwhea>cd server

C:\Users\dwhea\server>set FLASK_APP=app.py

C:\Users\dwhea\server>python -m flask run
 * Serving Flask app "app.py"
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [27/Apr/2020 20:19:33] "[37mHEAD / HTTP/1.1[0m" 200 -
127.0.0.1 - - [27/Apr/2020 20:19:33] "[37mGET / HTTP/1.1[0m" 200 -
127.0.0.1 - - [27/Apr/2020 20:19:35] "[37mGET / HTTP/1.1[0m" 200 -
127.0.0.1 - - [27/Apr/2020 20:19:35] "[33mGET /favicon.ico HTTP/1.1[0m" 404 -
```

By default, Flask will not reflect any changes made to the application. To do this we need to set the FLASK_DEBUG environment variable to 1 using the following command:

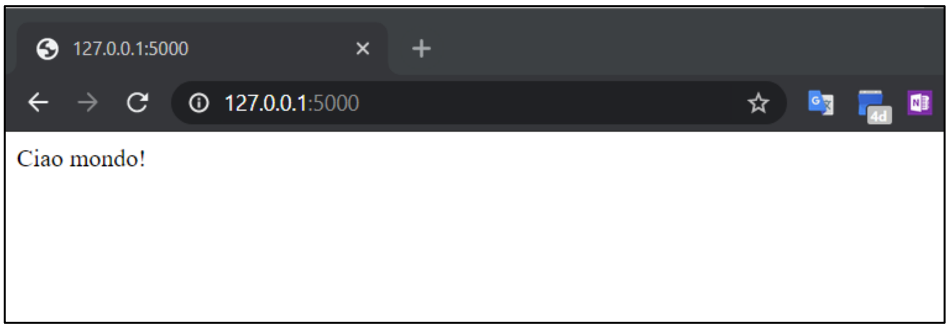
```
> set FLASK_DEBUG=1
```

Now if we make the following amendment to the app.py file and save it.



```python
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Ciao mondo!'
```

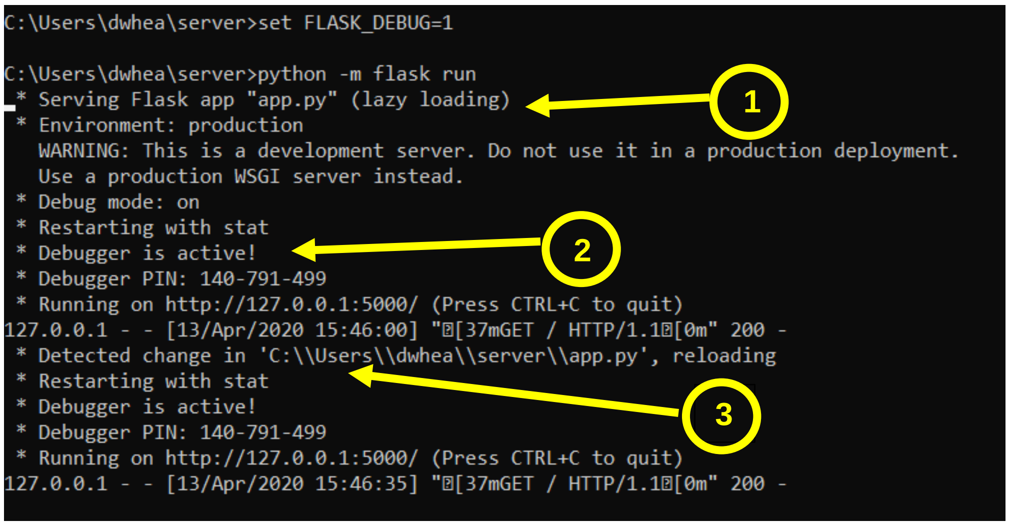
The change is reflected as soon as the web page is refreshed, as shown below.



Ciao mondo!

Referring to the command console again shows an updated report detailing the changes made. The three points of note are highlighted below:

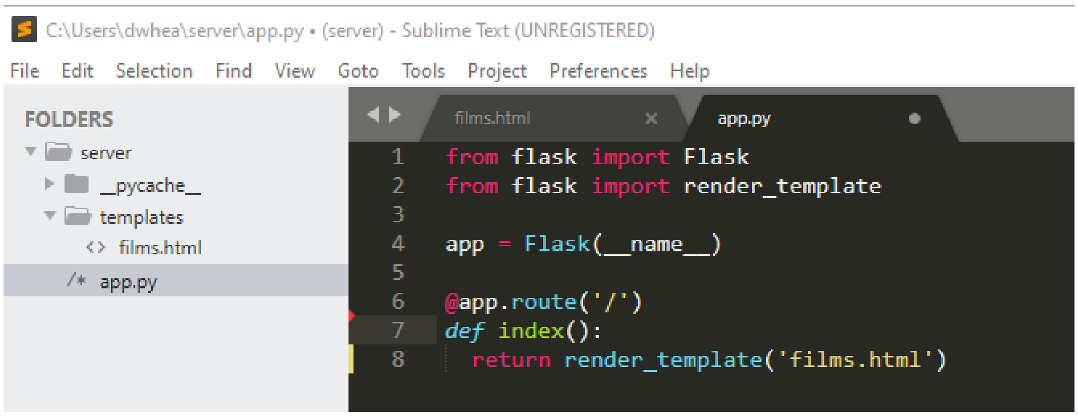
1. Lazy loading - which means that the elements of the web app are only loaded when required.
2. Debugger is now active.
3. After saving the app.py file the change is detected and the page reloaded automatically.



```
C:\Users\dwhea\server>set FLASK_DEBUG=1

C:\Users\dwhea\server>python -m flask run
 * Serving Flask app "app.py" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 140-791-499
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [13/Apr/2020 15:46:00] "[37mGET / HTTP/1.1[0m" 200 -
 * Detected change in 'C:\\Users\\dwhea\\server\\app.py', reloading
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 140-791-499
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [13/Apr/2020 15:46:35] "[37mGET / HTTP/1.1[0m" 200 -
```

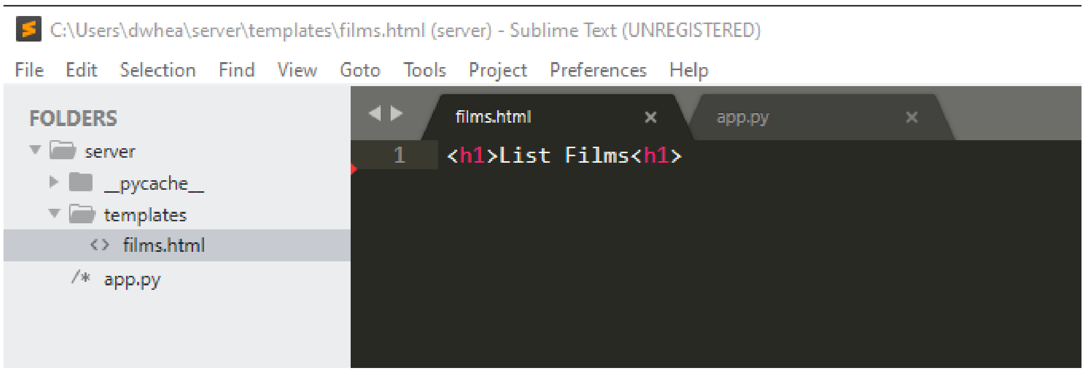
3.3 Adding a Template

Instead of returning a basic string, the render_template method allows a web page to be displayed. To do this the render_template method is explicitly imported, and the decorator amended to return the file 'films.html' as shown below.


C:\Users\dwhea\server\app.py • (server) - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help
FOLDERS
server
__pycache__
templates
films.html
app.py

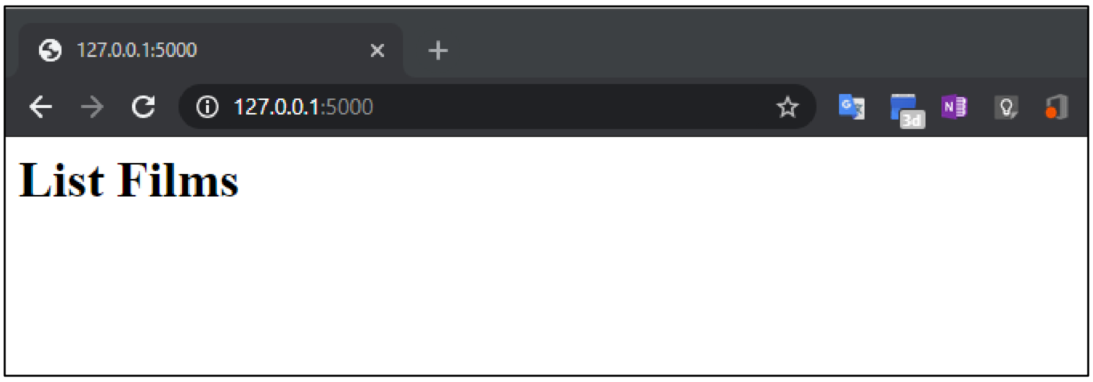
```
from flask import Flask
from flask import render_template

app = Flask(__name__)

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

In parallel we create a simple 'films.html' file and save it to a new sub folder called 'templates' in the server directory.


C:\Users\dwhea\server\templates\films.html (server) - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help
FOLDERS
server
__pycache__
templates
films.html
app.py

```
<h1>List Films<h1>
```

So long as flask is still running, refreshing the browser will now display the updated page.



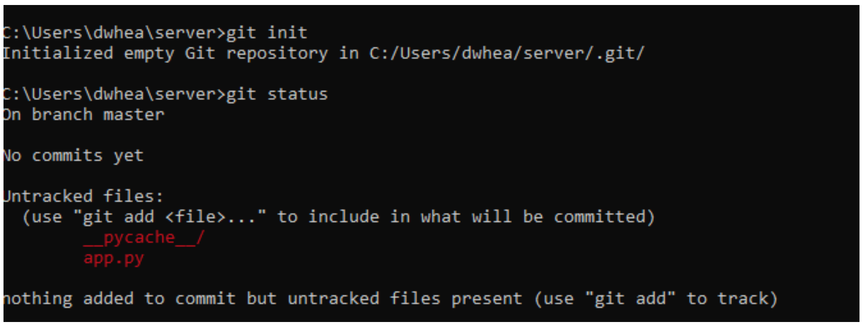
List Films

3.3 Setting up Git

Following the steps outlined in section 2.5 and assuming git has already been installed, the first step in the process is to initialise an empty repository using the following command.

> git init

Using the git status command, we see that our 'app.py' file is in the repository but not currently tracked.

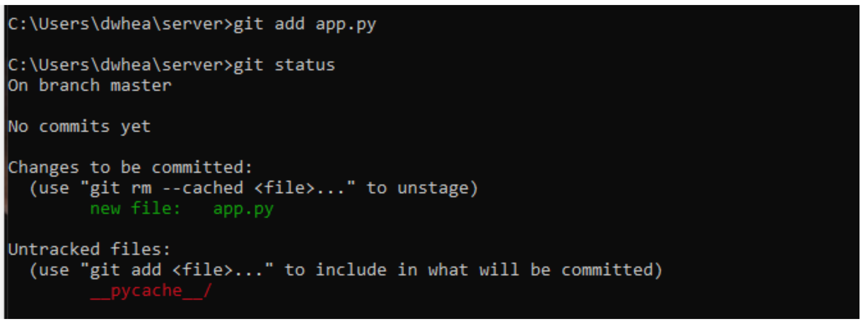


```
:\Users\dwhea\server>git init
Initialized empty Git repository in C:/Users/dwhea/server/.git/

:\Users\dwhea\server>git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        __pycache__/
        app.py

nothing added to commit but untracked files present (use "git add" to track)
```

To move to file to the staging area we use

> git add <filename>

Using the git status command again we can observe that our application is ready to be committed.



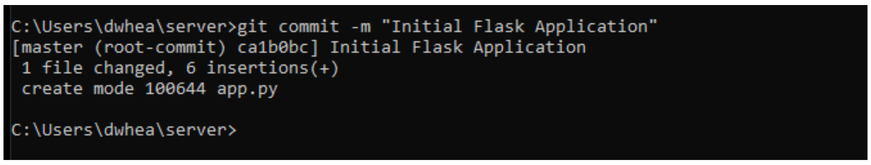
```
C:\Users\dwhea\server>git add app.py

C:\Users\dwhea\server>git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   app.py

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        __pycache__/
```

To commit our staging area to our local repository we then use

> git commit -m "<message>"

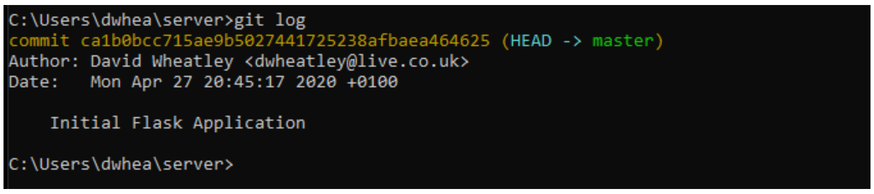
Where the message allows us to provide a brief explanation of the changes made, which can subsequently be viewed using the git log command.

If a longer more detailed message is required omitting the -m "message" will cause a text editor to be presented into which additional details can be added.



```
C:\Users\dwhea\server>git commit -m "Initial Flask Application"
[master (root-commit) ca1b0bc] Initial Flask Application
 1 file changed, 6 insertions(+)
 create mode 100644 app.py

C:\Users\dwhea\server>
```

The application has now been committed to the local staging area and reviewing the Git log reveals the commit reference ID along with the author, time and commit message.



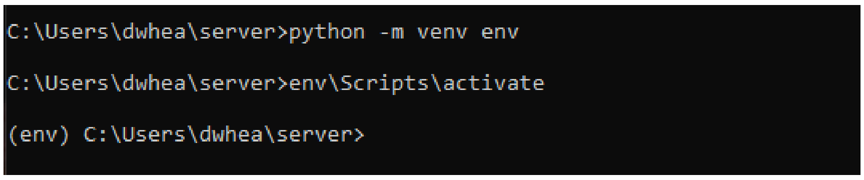
```
C:\Users\dwhea\server>git log
commit ca1b0bcc715ae9b5027441725238afbaea464625 (HEAD -> master)
Author: David Wheatley <dwheatley@live.co.uk>
Date:   Mon Apr 27 20:45:17 2020 +0100

    Initial Flask Application

C:\Users\dwhea\server>
```

4 Deploying to the Heroku platform

As describe in section 2.4 Heroku provides isolated environments, called containers, in which an application and the associated dependencies are hosted. We also noted that by creating representative environments locally we can avoid issues arising from working with different standards of dependencies.

4.1 Creating a local virtual environment

To create an equivalent isolated virtual environment on the local machine we use the following commands. Once the virtual environment has been created, we note the command prompt is prefixed with (env).



```
C:\Users\dwhea\server>python -m venv env

C:\Users\dwhea\server>env\Scripts\activate

(env) C:\Users\dwhea\server>
```

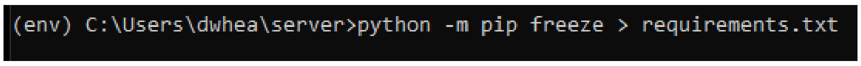
From within the virtual environment we can now re-install the required libraries, flask and gunicorn. Gunicorn is a web server capable of processing multiple request at a time and is required to support remote hosting on Heroku.

4.2 Configuring the remote environment

To configure the Heroku environment we need 3 files

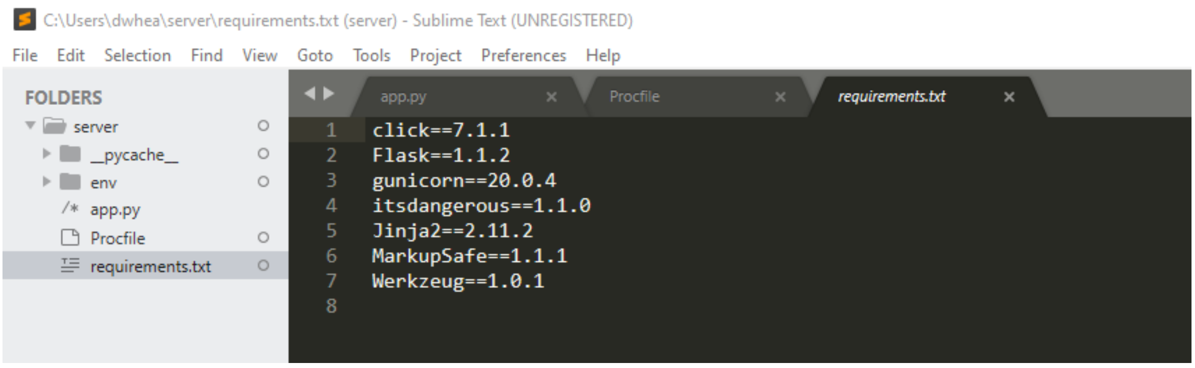
- Requirement.txt – Into which we will define the libraries that are required for our app to function.
- Profile - Which specifies how the app will be executed on start-up
- Runtime.txt - To specify which version of Python to use.

The easiest and most robust way to populate the requirements file is to copy the local virtual environment. This is achieved using the following command from within the virtual environment.



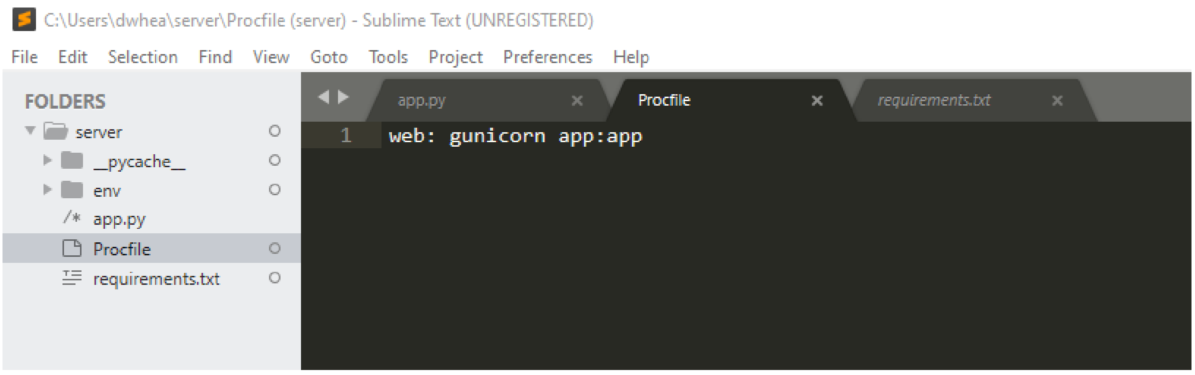
```
(env) C:\Users\dwhea\server>python -m pip freeze > requirements.txt
```

Once execute a complete requirements list is created and stored in the working directory. It is worth noting that this has automatically brought across the flask dependencies Werkzeug and Jinja2, as discussed in section 2.2.


C:\Users\dwhea\server\requirements.txt (server) - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help
FOLDERS
server
pycache
env
app.py
Procfile
requirements.txt
app.py
Procfile
requirements.txt

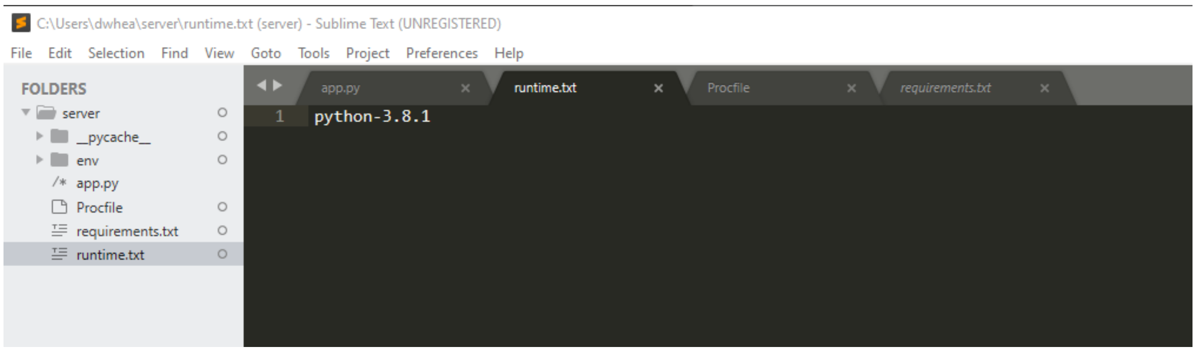
```
click==7.1.1
Flask==1.1.2
gunicorn==20.0.4
itsdangerous==1.1.0
Jinja2==2.11.2
MarkupSafe==1.1.1
Werkzeug==1.0.1
```

Saving the following Procfile to the working directory tells the server how to run our application


C:\Users\dwhea\server\Procfile (server) - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help
FOLDERS
server
pycache
env
app.py
Procfile
requirements.txt
app.py
Procfile
requirements.txt

```
web: gunicorn app:app
```

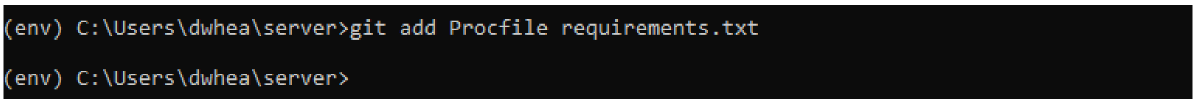
adding a runtime.txt file completes our configuration.


C:\Users\dwhea\server\runtime.txt (server) - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help
FOLDERS
server
pycache
env
app.py
Procfile
requirements.txt
runtime.txt
app.py
runtime.txt
Procfile
requirements.txt

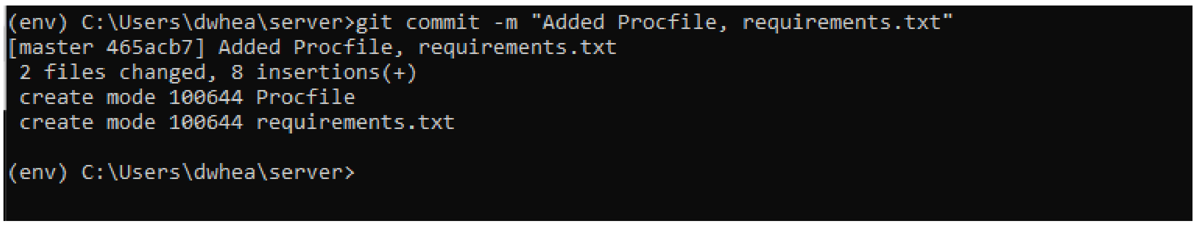
```
python-3.8.1
```

4.3 Pushing the app to Heroku

The configuration files created in 4.2 must now be added to the staging area and committed using the same commands detailed in section 3.3



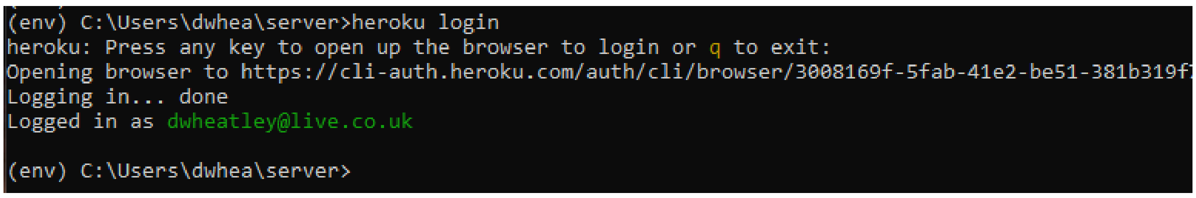
```
(env) C:\Users\dwhea\server>git add Procfile requirements.txt

(env) C:\Users\dwhea\server>
```



```
(env) C:\Users\dwhea\server>git commit -m "Added Procfile, requirements.txt"
[master 465acb7] Added Procfile, requirements.txt
 2 files changed, 8 insertions(+)
 create mode 100644 Procfile
 create mode 100644 requirements.txt

(env) C:\Users\dwhea\server>
```

Assuming Heroku has already been installed we can then login by typing

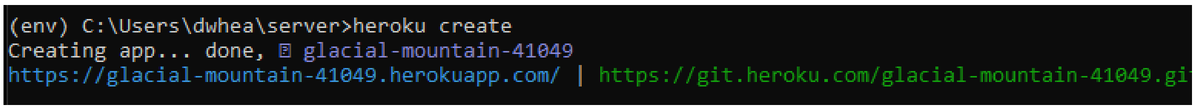
> heroku login



```
(env) C:\Users\dwhea\server>heroku login
heroku: Press any key to open up the browser to login or q to exit:
Opening browser to https://cli-auth.heroku.com/auth/cli/browser/3008169f-5fab-41e2-be51-381b319f
Logging in... done
Logged in as dwheatley@live.co.uk

(env) C:\Users\dwhea\server>
```

Followed by creating the remote repository, (https://glacial-mountain-41049.herokuapp.com/) using the command.

> heroku create

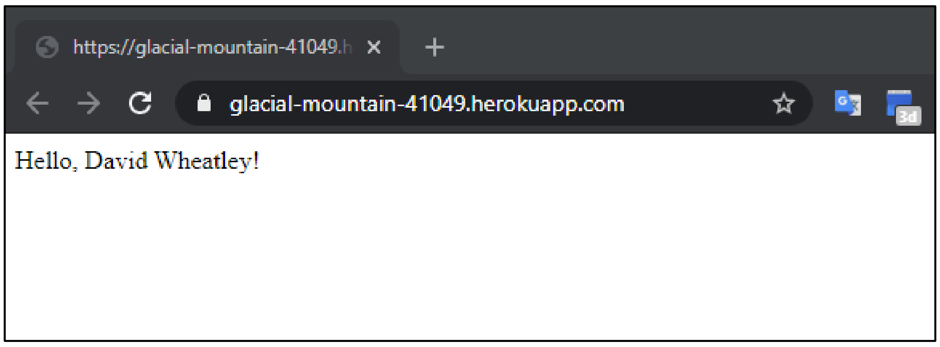


```
(env) C:\Users\dwhea\server>heroku create
Creating app... done, ⬢ glacial-mountain-41049
https://glacial-mountain-41049.herokuapp.com/ | https://git.heroku.com/glacial-mountain-41049.gi
```

The files can then be copied into the container by typing

> git push heroku master

After which the application is hosted on the Heroku server and available in any browser using the provided URL.



https://glacial-mountain-41049.h
glacial-mountain-41049.herokuapp.com

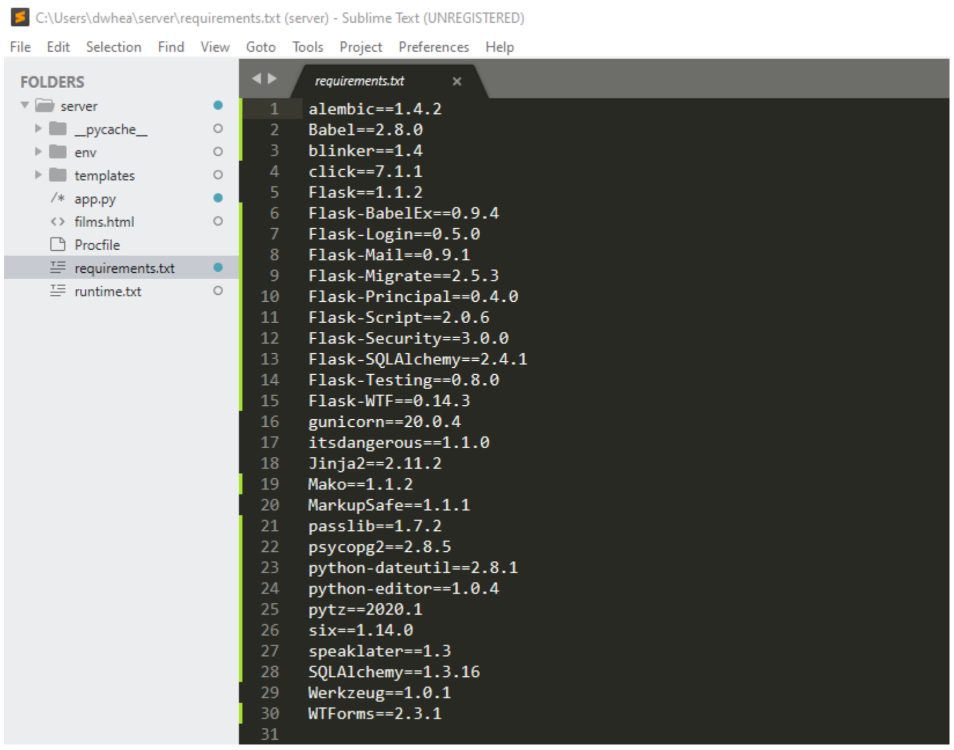
Hello, David Wheatley!

5 Connecting to a database

5.1 Updating the local environment

To enable our application to connect to a database using the SQLAlchemy ORM, the associated flask extensions (flask-script, flask-testing, flask-sqlalchemy, flask-migrate, flask_security) and the database engine (psycopg2) need to be installed.

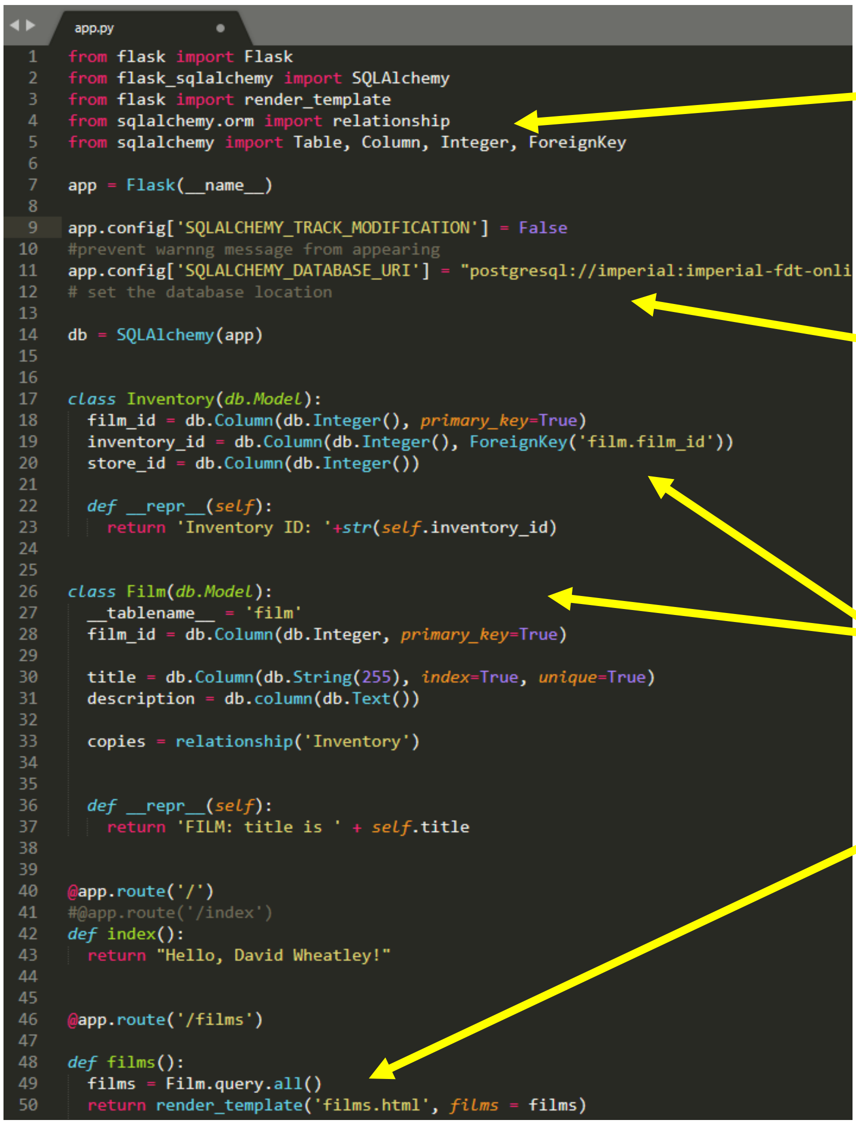
Re-running the pip freeze command, as in section 4.2, provides the following updated requirements.txt file.


C:\Users\dwhea\server\requirements.txt (server) - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help
FOLDERS
server
__pycache__
env
templates
app.py
films.html
Procfile
requirements.txt
runtime.txt
requirements.txt

```
alembic==1.4.2
Babel==2.8.0
blinker==1.4
click==7.1.1
Flask==1.1.2
Flask-BabelEx==0.9.4
Flask-Login==0.5.0
Flask-Mail==0.9.1
Flask-Migrate==2.5.3
Flask-Principal==0.4.0
Flask-Script==2.0.6
Flask-Security==3.0.0
Flask-SQLAlchemy==2.4.1
Flask-Testing==0.8.0
Flask-WTF==0.14.3
gunicorn==20.0.4
itsdangerous==1.1.0
Jinja2==2.11.2
Mako==1.1.2
MarkupSafe==1.1.1
passlib==1.7.2
psycopg2==2.8.5
python-dateutil==2.8.1
python-editor==1.0.4
pytz==2020.1
six==1.14.0
speaklater==1.3
SQLAlchemy==1.3.16
Werkzeug==1.0.1
WTForms==2.3.1
```

5.2 Modifying the application

The app.py application code now needs to be changed to connect to the database and define the classes into which the database entities will map as well as to pass the data to be rendered onto a web page.



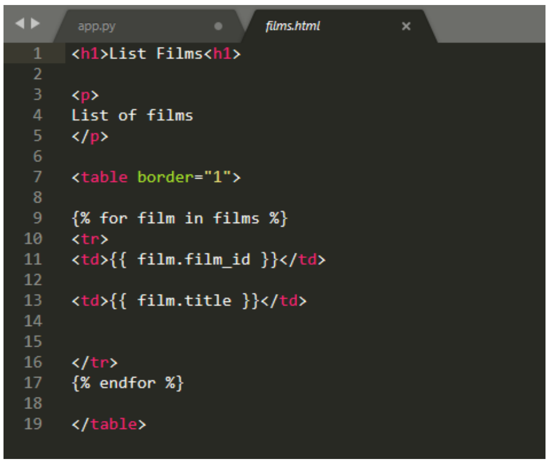
```python
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask import render_template
from sqlalchemy.orm import relationship
from sqlalchemy import Table, Column, Integer, ForeignKey

app = Flask(__name__)

app.config['SQLALCHEMY_TRACK_MODIFICATION'] = False
#prevent warnng message from appearing
app.config['SQLALCHEMY_DATABASE_URI'] = "postgresql://imperial:imperial-fdt-onli
# set the database location

db = SQLAlchemy(app)


class Inventory(db.Model):
  film_id = db.Column(db.Integer(), primary_key=True)
  inventory_id = db.Column(db.Integer(), ForeignKey('film.film_id'))
  store_id = db.Column(db.Integer())

  def __repr__(self):
    return 'Inventory ID: '+str(self.inventory_id)


class Film(db.Model):
  __tablename__ = 'film'
  film_id = db.Column(db.Integer, primary_key=True)

  title = db.Column(db.String(255), index=True, unique=True)
  description = db.column(db.Text())

  copies = relationship('Inventory')


  def __repr__(self):
    return 'FILM: title is ' + self.title


@app.route('/')
#@app.route('/index')
def index():
  return "Hello, David Wheatley!"


@app.route('/films')

def films():
  films = Film.query.all()
  return render_template('films.html', films = films)
```

1. Import the additional libraries to be used by the application
2. Configure the database connection ensuring the correct URI format is used
3. Create the classes that our database will map to.
4. Modify the route() decorator to define the variable films and pass to the render_template method

Within the classes (point 3) each attribute in the data tables along with various attributes that the ORM will help to maintain, e.g. keys and index values.

Having connected to the database we now need to define how it is displayed. From our application code above the decorator function (point 4) now passes the variable films makes a call to the render_template method.

The next step is to modify the 'films.html' file to process this data and populate a table on the webpage with a rendered string as shown below.

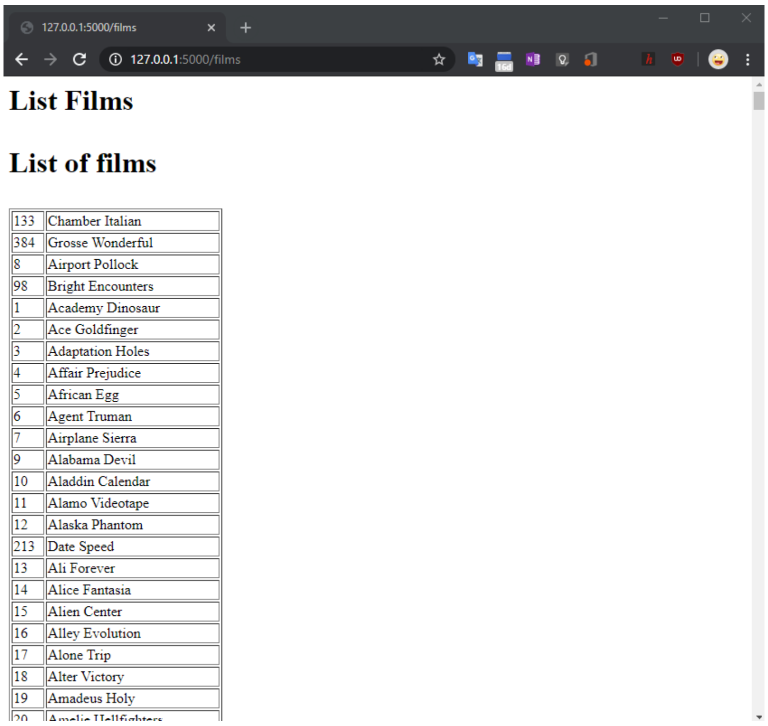


```
<h1>List Films<h1>

<p>
List of films
</p>

<table border="1">

{% for film in films %}
<tr>
<td>{{ film.film_id }}</td>

<td>{{ film.title }}</td>


</tr>
{% endfor %}

</table>
```

- <table border="1">
  Creates a basic table
- {% for film in films%}
  Loops through every entry in films
- <tr>
  Creates a row in the table for each object
- <td>
  Defines the 2 columns (cells) within a row and populates them with the id and title from films.

5.3 Running the updated application locally

Opening the browser with the URL http://127.0.0.1:5000/films now reveals a fully populated table.



List Films

List of films

133	Chamber Italian
384	Grosse Wonderful
8	Airport Pollock
98	Bright Encounters
1	Academy Dinosaur
2	Ace Goldfinger
3	Adaptation Holes
4	Affair Prejudice
5	African Egg
6	Agent Truman
7	Airplane Sierra
9	Alabama Devil
10	Aladdin Calendar
11	Alamo Videotape
12	Alaska Phantom
213	Date Speed
13	Ali Forever
14	Alice Fantasia
15	Alien Center
16	Alley Evolution
17	Alone Trip
18	Alter Victory
19	Amadeus Holy
20	Amelie Hellfighters

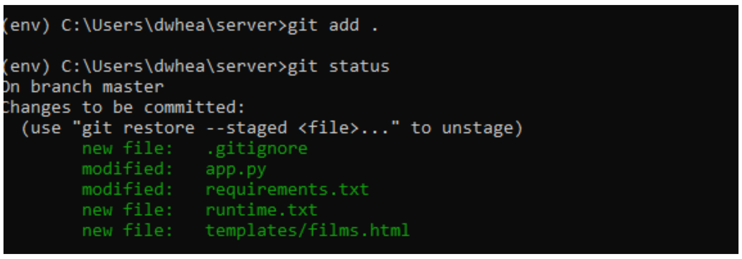
5.4 Hosting the updated application on Heroku

To host the updated application on Heroku the process used in section 4.2 is repeated. The only difference is that we now have many more files to stage and it is therefore more convenient to use a following command to stage all the files in the working directory.

> > git add .

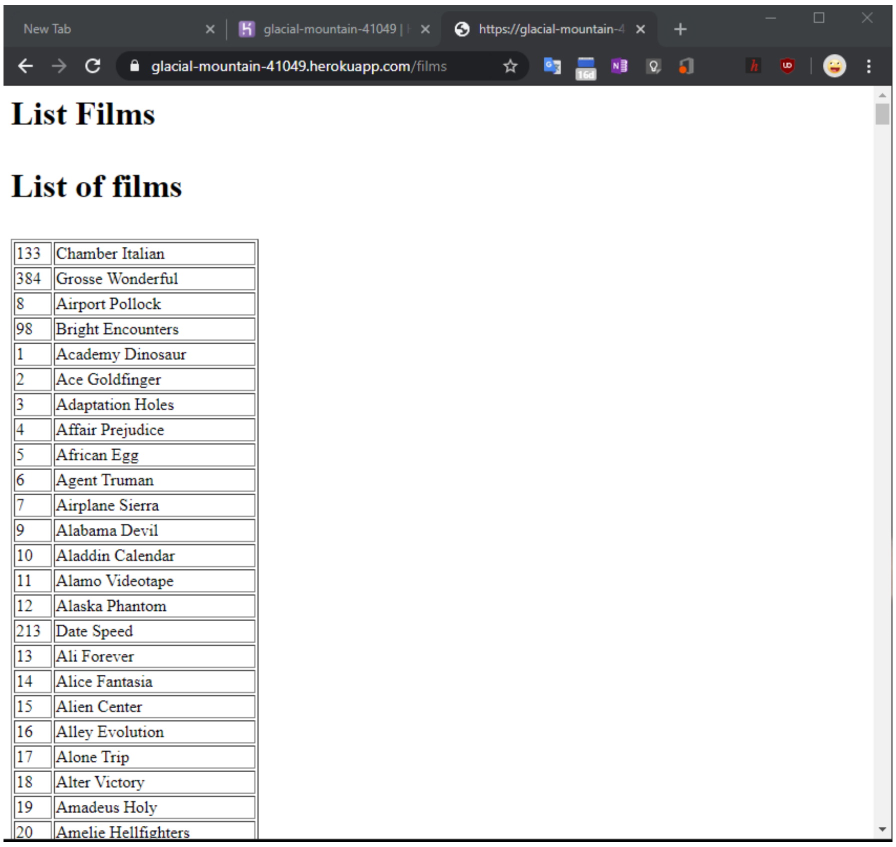
Saving a file called '.gitignore' in the working directory defines those files which should be excluded from a bulk staging, such as the virtual environment and git repository setup files. Standard '.gitignore' files can easily be downloaded from GitHub.

Once the '.gitignore' file is in place the 'git add .' command can be run and the status checked to ensure everything has been successfully updated.



```
env) C:\Users\dwhea\server>git add .

env) C:\Users\dwhea\server>git status
n branch master
hanges to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   .gitignore
        modified:   app.py
        modified:   requirements.txt
        new file:   runtime.txt
        new file:   templates/films.html
```

These can now be committed and pushed to the Heroku repository, as defined in section 4.3, after which the updated app can be viewed by connecting to the https://glacial-mountain-41049.herokuapp.com/films URL in a web browser.


New Tab
glacial-mountain-41049 |
https://glacial-mountain-4
glacial-mountain-41049.herokuapp.com/films

List Films

List of films

133	Chamber Italian
384	Grosse Wonderful
8	Airport Pollock
98	Bright Encounters
1	Academy Dinosaur
2	Ace Goldfinger
3	Adaptation Holes
4	Affair Prejudice
5	African Egg
6	Agent Truman
7	Airplane Sierra
9	Alabama Devil
10	Aladdin Calendar
11	Alamo Videotape
12	Alaska Phantom
213	Date Speed
13	Ali Forever
14	Alice Fantasia
15	Alien Center
16	Alley Evolution
17	Alone Trip
18	Alter Victory
19	Amadeus Holy
20	Amelie Hellfighters

6 Conclusion

Having successfully created a template file connecting to the dvdrental database and hosted it on the Heroku platform the foundations have been put in place. This could now be enhanced through developing more complex queries, using SQLAlchemy, or utilising CSS style sheets to improve the appearance of the application.