Flask

Flask is a micro web framework for python and based on the Werkzeug toolkit and Jinja2 template engine. The framework exists as on open source software under a BSD license. Due to its light weight in terms of size, Flask does not require any tools or libraries, has no database abstraction layer or form validation, however supports extensions that implement features that feel as though they were implemented in Flask itself.

Python Control File

The python (.py) control file is found in the base folder labeled TestPython.py. This file initialises the website to a local host and contains defined links and their designated scripts.

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

#User defined Funtions

if __name__ == “__main__”:
      app.run()
```

Image 1: Initialising Flask

Image 1 above provides a basic initialisation of Flask after it is installed on a system. Running this python code would create a empty Flask application under a local host.

```
@app.route('/boot')
@login_required
Def boot_test():
        return render_template('TestBoot.html')
```

Image 2: User defined function

Image 2 show a user defined function and is the function used to display the post-login page of the provided example application. The first line defines the web route, the second is a function decorator that will be explained below. The defined function on line 3 renders the HTML file specified and displays it when the user is at the specified route.

```
def login_required(f):
        @wraps(f)
        def wrap(*args, **kwargs):
                if 'logged_in' in session:
                        return f(*args, **kwargs)
                else:
                        return redirect(url_for('login'))
                return wrap
```

Image 3: Function decorator – login_required

The following function in Image 3 uses the Flask extension session and a functool called wraps, it is a function decorator that can be put above any user defined function as shown in Image 2. This decorator checks to see if the user is logged in before allowing access to any routes with the decorator specified before them. If the user is not logged in or if the cookie expires, the user is returned to the login page.

```
@app.route('/', methods=['GET, 'POST'])
Def login():
        Error = None
        If request.method == 'POST'
                If request.form['username'] != 'admin' or
request.form['password'] != 'admin':
                        Error = 'Invalid Credentials. Please try again.'
                Else:
                        Session['logged_in'] = True
                        Return redirect(url_for('boot_test'))
        Return render_template('login.html', error=error)
```

Image 4: Login function

The login function is the first route Flask loads as it is specified under the app.route('/'), the base route. The function connects to a html form defined in the login.html file that when filled out and posted, makes the function check for correct user Credentials before either allowing or denying access to the rest of the web application.

File Structure

Flask follows a simple file strucuture where the base python files exist in the root folder. Any HTML files exist in a 'template' folder and any javascript or CSS files exist in a 'static' folder. Below is a sample image of an example file directory that is compatible with Flask.

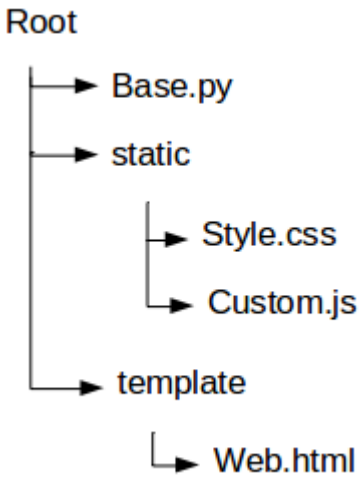

Image 5: Sample Flask file structure

Psycopg2

Psycopg2 is a Flask extension that acts as a postgresql adapter for the Flask application. It provides a secure access to any specified postgresql database running on the system. It is import like any other Flask extension using import psycopg2 and uses SQL statements to access tuples in the linked database. Note: psycopg2 must be installed on the system like any other Flask extention before used.

```
def connect_db():
        return psycopg2.connect(dbname=”postgres”, user=”postgres”,
password=”root”, port=5432)

@app.before_request
def before_request():
        g.db = connect_db()
```

Image 6: Database connection

The connect_db function in Image 6 returns a database connection which is called in the following function. @app.before_request is a special function as it is called before any user requests are made and therefore opens up a database connection before the user is allowed to do anything. The 'g' variable is a special Flask variable that stores the database connection and is also used in user database requests.

```
@app.teardown_request
def teardown_request(exception):
        db = getattr(g, 'db', None)
        if db is not None:
                db.close()
```

Image 7: Database connection closed

Similarly to @app.before_request, @app.teardown_request executes after a user request finishes. This defined function takes any exceptions and makes sure that the connection to the postgresql database is closed.

```
@app.route('/add', methods=['GET', 'POST'])
@login_required
def add_entry():
	cursor = g.db.cursor()
	cur = cursor.execute('SELECT * FROM entries ORDER BY id DESC;')
	entries=[dict(id=row[0], title=row[1], text=row[2]) for row in
cursor.fetchall()]
	if request.method == 'POST':
		if request.form['title']:
			cursor.execute('INSERT INTO entries (title, text) VALUES
(%s, %s)', (request.form['title'], request.form['text']))
			g.db.commit()
			return redirect(url_for('show_entries'))

	return render_template('new.html', entries = entries)
```

Image 8: Add tuple function

Image 8 shows the entire add tuple function. This function uses psycopg2 to first create a user cursor that allows for SQL commands to be executed in the python code. The first SQL statement is a SELECT statement used to grab all the current tuples and put the in a dict variable type, used later to be displayed on the screen to the user. If a POST request is sent from the form in the html file, and only if the 'title' input is not empty, the cursor executes an INSERT command adding the newly created tuple to the database. Unlike the SELECT command, the INSERT command changes the database and therefore requires to be committed so that the database knows to finalise the change and thus `g.db.commit()` is executed. Very similar actions are taken with the Update and Delete functions with only SQL statements changing.

BootStrap

Flask integrates Bootstrap quite easily, flask_bootstrap needs to be installed on the system and included into the python document. To access Bootstrap, it's CSS and HTML files must be referenced in the HTML files that would like to use them. `Bootstrap(app)` must also be included in the python document and can be found just under `app = Flask(__name__)` in the provided files.

```
{% extends “bootstrap/base.html %}

{% block head %}

{% block style %}
	{{super()}}
	<link rel=”stylesheet” type=”text/css” href=”{{ url_for('static', filename='bootstrap.css') }}”>
	<link rel=”stylesheet” href=”{{ url_for('static', filename='bootstrap.min.css') }}” media=”screen”>
{% endblock style %}

{% endblock head %}
```

Image 9: Bootstrap in HTML

The above code shows how to link Bootstrap in HTML using Flask, everything done in child templates is based on blocks, this is used primarily to save typing effort. The Jinja2 super() function is a powerful function that allows for amending blocks instead of replacing them. Most Bootstrap styles are accessed via class referencing which can be done on <tags> at an individual level.

Available blocks

Block name	Outer Block	Purpose
doc		Outermost block.
html	doc	Contains the complete content of the `<html>` tag.
html_attribs	doc	Attributes for the HTML tag.
head	doc	Contains the complete content of the `<head>` tag.
body	doc	Contains the complete content of the `<body>` tag.
body_attribs	body	Attributes for the Body Tag.
title	head	Contains the complete content of the `<title>` tag.
styles	head	Contains all CSS style `<link>` tags inside head.
metas	head	Contains all `<meta>` tags inside head.
navbar	body	An empty block directly above *content*.
content	body	Convenience block inside the body. Put stuff here.
scripts	body	Contains all `<script>` tags at the end of the body.

Image 10: Available blocks in Bootstrap