Development report for Python Flask CallBack dating application

Tom Brown

Edinburgh Napier University SOC

Module: Advanced Web Tech CW2

Tutor: Simon Wells

Date: November 30th 2016

Summary

This report will illustrate the design, development, architecture and processes during creation of a Python Flask web application. Flask is a micro web framework written in Python, an object-oriented, programming language.

Furthermore, the document will evaluate the application as well as gauge any personal achievements and hurdles overcome during the operation. Findings will be noted and any potential enhancements to the application will be highlighted.

Table of Contents

1. Introduction

To expand on the skills that were learned for CW1, a dating website, similar in concept to Plenty of Fish or Match would be constructed. The application is named CallBack. It will allow users to create an account and register on two different pricing tiers. One basic tier, which is free of charge and a premium payer that costs ten pound. At the outset the plan was to introduce Stripe API integration for payment handling but this was not possible inside the Levinux environment.

After signing up to a free or basic account, users can create a profile and upload an avatar for their account. They can also update these details at a later date. Users can log in and out of their account to perform admin. Users can also browse other profiles created within the application.

2. Web architecture

2.1 ReST

REST (Representational State Transfer), was first introduced and outlined by Roy Fielding in 2000 [1]. CallBack is a ReSTful application, built using the Flask micro-framework for Python. REST architecture is built on top of the HTTP (Hyper Text Transfer Protocol) protocol that the web implements. REST utilizes HTTP verbs. HTTP verbs are methods that allow requests to be made to a web server that can make requests to GET, POST, PATCH, PUT and DELETE content from the server. When working with a datastore, these verbs correspond with the CRUD actions of create, read, update and delete. In a ReSTful Flask API (Application Programming Interface), the most common are the GET request. GET is the most common due to it being the main method for serving web pages. REST architecture is stateless, meaning that data from the client is not stored on the server. The status of the session is held by the client in the form of a cookie. Being stateless is crucial to maintaining scalability as it allows concurrent users to make requests to the server [2] without impacting performance. However, it doesn't have the built-in error handling of its main competitor, SOAP (Simple Object Access Protocol). SOAP also relies completely on XML (eXtensible Markup Language) [3], meaning that some responses and requests can be complex and manually built [4]. Nonetheless, REST APIs simplicity, speed its lightness and extensibility mean it is the prevalent architecture.

3. Enhancements

Functionality-wise, the application does expand on the submission for CW1. However, there could be some improvements made. With this in mind, a few points are outlined below.

3.1 Geocoding

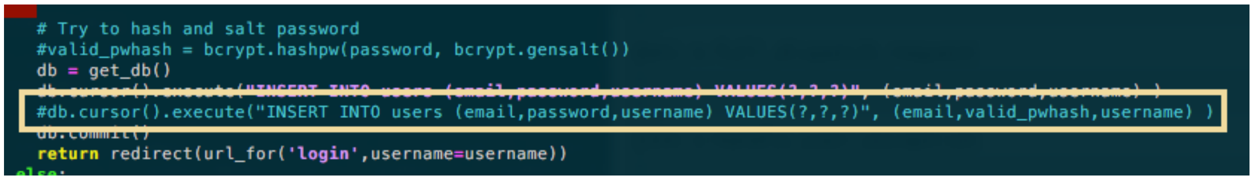
On the Individual profile pages, the Google Maps API [5] was implemented to show a map. Maps API was intended to display the location of the user but was proving difficult as the map is plotted using a latitude and longitude. The create profile form takes in a place as one of its fields as a place name, i.e., Glasgow. However, it was proving too difficult regarding knowledge and time to find a way to connect this to the Geocoding element of Google maps.

3.2 Improved interface design

Time-permitting, the overall look, and feel of the application could be enhanced with some resources allotted to interface design. The Bootstrap CSS [6] provides a clean, professional look. Nonetheless, it would be of benefit to add some more custom styles over and above what is already included going forward, to focus aesthetics for its target demographic.

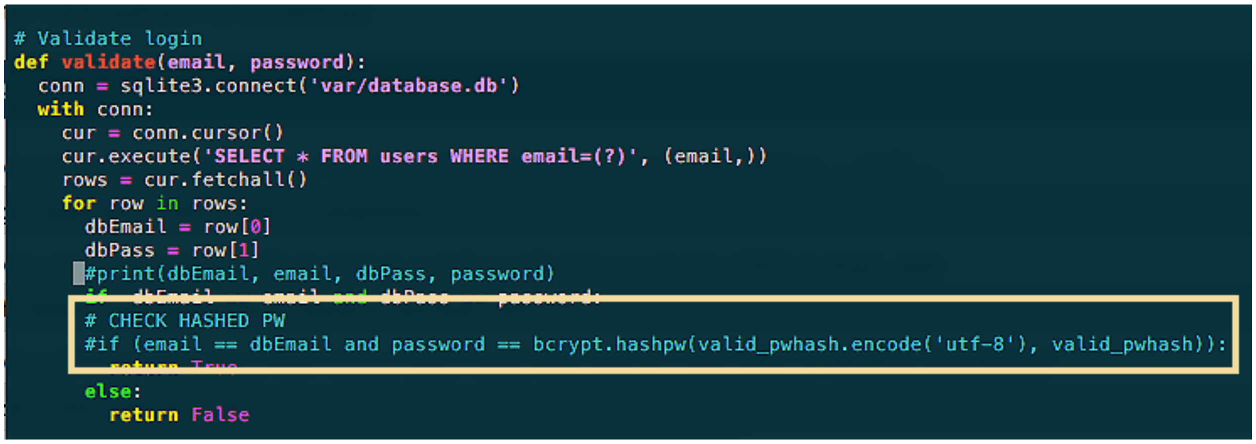
3.3 User authentication hashing

The Bcrypt password salting and hashing module was imported into the app, but in the end, the functionality had to be removed. During ongoing testing, via the Python Shell, it was evidenced that the hashed passwords were being stored in the sqlite3 database. However, any attempt at logging was failing and the only way to rectify it was to remove bcrypt. The code has been left in the app but has been commented out. After some time and no success in fixing the issue, it was decided to revert to the previous authentication functionality. It would be worthy to return to this in the future to implement bcrypt.
See images Below.



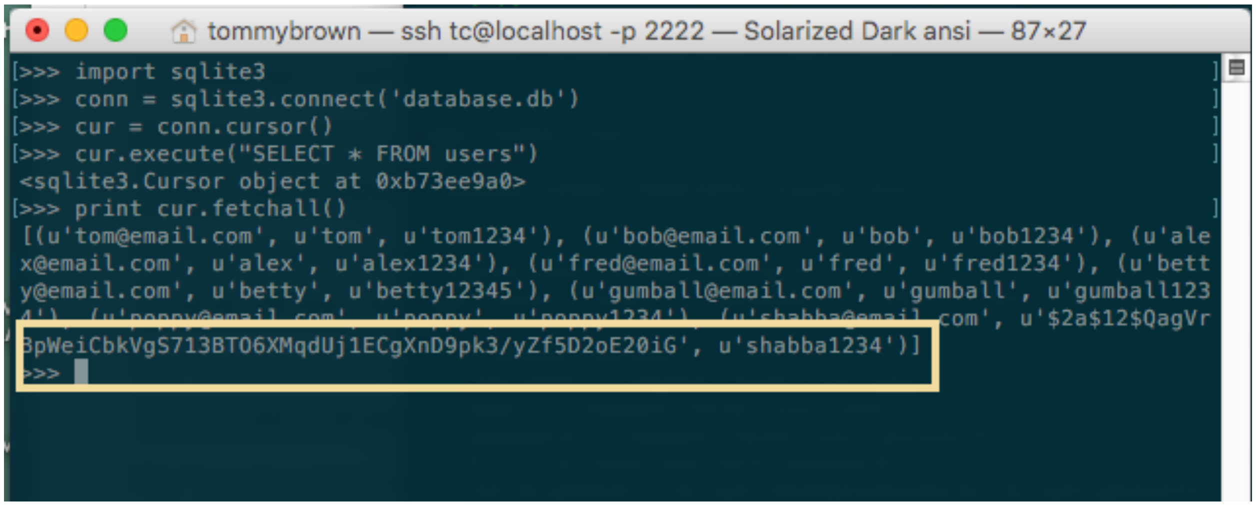
```
  # Try to hash and salt password
  #valid_pwhash = bcrypt.hashpw(password, bcrypt.gensalt())
  db = get_db()
  db.cursor().execute("INSERT INTO users (email,password,username) VALUES(?,?,?)", (email,password,username) )
  #db.cursor().execute("INSERT INTO users (email,password,username) VALUES(?,?,?)", (email,valid_pwhash,username) )
  db.commit()
  return redirect(url_for('login',username=username))
else:
```

Figure 1: Attempt at password hashing.



```
# Validate login
def validate(email, password):
  conn = sqlite3.connect('var/database.db')
  with conn:
    cur = conn.cursor()
    cur.execute('SELECT * FROM users WHERE email=(?)', (email,))
    rows = cur.fetchall()
    for row in rows:
      dbEmail = row[0]
      dbPass = row[1]
      #print(dbEmail, email, dbPass, password)
      if dbEmail == email and dbPass == password:
      # CHECK HASHED PW
      #if (email == dbEmail and password == bcrypt.hashpw(valid_pwhash.encode('utf-8'), valid_pwhash)):
        return True
      else:
        return False
```

Figure 2: Shows code for attempt to use bcrypt module.



```
tommybrown — ssh tc@localhost -p 2222 — Solarized Dark ansi — 87×27
>>> import sqlite3
>>> conn = sqlite3.connect('database.db')
>>> cur = conn.cursor()
>>> cur.execute("SELECT * FROM users")
<sqlite3.Cursor object at 0xb73ee9a0>
>>> print cur.fetchall()
[(u'tom@email.com', u'tom', u'tom1234'), (u'bob@email.com', u'bob', u'bob1234'), (u'ale
x@email.com', u'alex', u'alex1234'), (u'fred@email.com', u'fred', u'fred1234'), (u'bett
y@email.com', u'betty', u'betty12345'), (u'gumball@email.com', u'gumball', u'gumball123
4'), (u'poppy@email.com', u'poppy', u'poppy1234'), (u'shabba@email.com', u'$2a$12$QagVr
3pWeiCbkVgS713BTO6XMqdUj1ECgXnD9pk3/yZf5D2oE20iG', u'shabba1234')]
>>> 
```

Figure 3: Shows password being hashed in the database.

3.4 Config File

A simple config file, based on the version in the workbook was attempted, yet it had to be discarded as it was throwing a key error for ip_address. In contrast, the same code worked fine in the example that was built during the labs for making a simple config file. After spending a significant amount of time trying to fix it, the decision was taken to push on and finish building the application without it.

3.5 Template logic

Some of the pages of the application can be very similar, save for some small pieces of the interface. For example, on the view profiles page, when a user is logged in, there is no need to have a ‘log in’ button at the end of the results. On the other hand, the button should be visible to anyone who is viewing the profiles just as a casual non-member or browser. Time was allotted to look into it but the build was almost complete, and it was proving difficult to fix with limited time.

3.6 Stripe payment API

At the outset of the project, the intention was to offer a two-tier registration. One that required payment [7], and one free of charge service. This was not possible working in the Levinux environment. It would be a worthwhile addition to the project to add this functionality in the future.

4. Critical Evaluation

In the project brief, it was specified that the web-app should ‘demonstrate your mastery of aspects of Flask covered so far. You should make appropriate use of some of the following: Routing (and URL hierarchy design), static files, requests, redirects, responses, templates, sessions, logging, testing, CSS, JavaScript, multiple users, and data storage.'
This section provides an analysis and evaluation of those requirements. You can find the GitHub repo at:

https://github.com/Tommy-B-Git/tb-project

4.1 General requirements

As is evidenced in the application codebase, many of the above requirements have been addressed. The application tries to replicate some of the features of popular dating websites such as Plenty of Fish, or Match.com [8]. In the creation of functionality for user registration, user authentication, and profile creation, many of the requirements have been satisfied. A significant number of routes were included, many of which handled both GET and POST requests. A meaningful URL hierarchy was built. However, some of the URLs were untidy due to variables being passed with a redirect, rather than with render_template; meaning that a user’s email is exposed in the URL which is not great for security. Static files were included, featuring custom CSS and an image directory for storing images. The paths to the images are held in the data store which is, on this occasion, sqlite3. The sqlite3 database has three tables, one for free and premium registrations and a third for all profile information and images.
A substantial number of templates were created in the creation of the app to enable a realistic looking clone of similar services that are already available. That said, some of the templates could be more amalgamated, discarded or streamlined with some better planning of the view functions. For example, if a form was going to be rendered on a true or false condition, some more logic to decide which form should be shown and embedded in the base.html, rather than having an array of templates for every eventuality. Nonetheless, the application works well with the templates used. Multiple users can access the application and their details for logging in to

their profiles held in the database. The details can be updated if desired. A simple testing script was included.

The code does not adhere strictly enough to DRY (don't repeat yourself) [9] principles. There is some code that given more time could have been refactored, for example, there are two validation functions for each tier of authentication. Going forward, this could be written into one function with more logic. Breaking the app down into more manageable modules would also have been of benefit, rather than one huge app.py file

5. Personal Evaluation

5.1 What I have learned

As the application grew in size, it became clear that some time spent at the outset, sketching out the application on paper and looking at the routes and requests would have been of benefit. To that end, more focus on time management would be helpful. It is apparent that confidence has been gained using the Python Flask framework. Lessons were learned that regular commits to the Git repository are essential. Notably, as an application gets bigger, the scope to inadvertently make a typo, or small hard-to-find error can cause problems for developers. Therefore, the ability to return to a working version of the code is crucial.

5.2 Challenges

There were many challenges. Some ended in failure as the focus had to be on a deliverable that was functioning and implementing as many of the features already mentioned. However, time was a crucial factor in deciding how much time could be allotted to specific areas where issues arose. Getting the password hashing properly working would have added value. The Google map to show the users location based on the form input to their profile would also have made for a better application and experience for the user. Regarding the code; to be able to break the application down into a more modular, manageable structure would be helpful for others should they need to work on it. Levinux limits what can be done for the scope of this project, and this in itself can present challenges in terms of data storage and some modules that cannot be imported. For example, MongoDB and Stripe payment integration. The reference material available tends to be for development environments other than Levinux.

6. Summary of Resources

The main resources utilized in the development of the CallBack application were as follows. The Flask micro framework and the Python programming language. Bootstrap was implemented in the creation of the program. Bootstrap is an open-source, front-end framework that allows for easy styling of web applications enabling rapid prototyping of responsive web applications. Google fonts API is a way of adding custom typefaces to your application out with the system fonts on your server. Google Maps was also implemented as referenced earlier.

7. References

1. http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

2. http://stackoverflow.com/questions/3105296/if-rest-applications-are-supposed-to-be-stateless-how-do-you-manage-sessions

3. https://www.tutorialspoint.com/soap/what_is_soap.htm

4. http://blog.smartbear.com/apis/understanding-soap-and-rest-basics/

5. https://developers.google.com/maps/

6. http://getbootstrap.com/

7. https://stripe.com/docs/checkout/tutorial

8. https://uk.match.com

9. https://code.tutsplus.com/tutorials/3-key-software-principles-you-must-understand--net-25161