UNIVERSITY OF HERTFORDSHIRE

Department of Computer Science

BSc Honours in Computer Science (Software Engineering)

6COM1053 – Computer Science Project

Final Report

April 2022

Welwyn Garden City Cricket Club (WGCCC) – Web-Based Payments & Notifications System

Joshua Molyneux

18043053

Supervised by: Dr. Christopher Baker

Table of Contents

Table of Contents (cont.)

List of Tables

Table of Figures

Table of Figures (cont.)

Joshua Molyneux
18043053

Abstract

Undertaking this project investigates the possibility and perhaps the necessity for automated payments at Welwyn Garden City Cricket Club (WGCCC) to help alleviate the manual steps needed to process them as per the current system. The project explores research in the form of a literature review necessary for backing the project hypothesis and the prospect of success, not necessarily in the form of a working artefact, but the conclusions that can be drawn from its experimentation.

Software design for the project includes producing design materials such as storyboards, user requirements, and database design that will assist in the implementation of the artefact. Implementation, based upon these design materials, will ultimately allow it to form results and conclusions based upon the approach to development.

The conclusions found the project to be successful and a working artefact was produced as a result. With changes and future considerations, the system can be improved, and the original idea of this project can be tested further if necessary.

Acknowledgements

Firstly, I would like to thank my supervisor for helping me whenever I needed it and offering good guidance.

I would also like to thank my peers for giving me motivational support throughout the project. Their presence was invaluable.

Finally, a thank you to my family for being there to emotionally support me if I needed it. They have done a great job getting me to this stage in my life and have given me everything.

1.0 Introduction

Inspiration for this project comes from a conversation I had with a family member who is a Club Official for Welwyn Garden City Cricket Club (WGCCC). He told me that he finds difficulty in managing Memberships with the current system in place as it is only done manually through email. Having played for WGCCC as a junior and knowing many people who play and work with the club, I feel like I want to contribute something back to the club community in a way I know how and in a way that is relevant to my prospects in the industry as a Software Engineer.

The main aim of this project is to develop a solution that allows Club Officials to manage Member payments for various types of Subscriptions in a centralised format using a data management system and a secure back-end web-based technology with an easy-to-use interface design. The system should utilise various security features to protect sensitive data belonging to members, such as, but not limited to, hashing of data and password-protected areas.

The solution should also notify Members through SMS and Email about forthcoming payments and subscription reminders. This notification system can also be used for various other problems in the future.

The process of the project and this report involves preliminary research in the form of a Literature Review, a Project Plan, Design plans and materials, the implementation process and finally a section to draw conclusions on the outcome of the project. This will include a self-evaluation and an extensibility review.

2.0 Literature Review

The Literature Review aims to outline and explore a collation of research around various topics associated with the project and processes that will be required for a suitable artefact to be produced from my findings. Conducting research for this review, it's vital for me to gain an understanding on how the current system works, but also any underlying software technologies that I may have to use for my project. This includes, but is not limited to, suitable programming languages, third-party payment gateways/providers, notification handlers & database management systems.

2.1 Current System

When improving or altering a system, it is important that I examine and investigate how the current system works to gain an understanding on what I am dealing with and how my proposed solution would integrate or change how the system works in any way.

Research completed for this problem included an initial personal communication with Christopher Molyneux, on the 1st of October 2021, who stated that the current system was not sufficient for his requirements and that a system that could help him easily manage subscriptions and payments would add efficiency to his role of Club Captain.

The current system is manual. Upon a prospective member entering the website, they are greeted with a 'New Players' section on the landing page. This shows information about Senior and Junior cricket teams handled by two different people respectively, including Christopher Molyneux above, who handles Senior subscriptions, and an Official named Russell Haggar. Currently, you must email either one of these Club Officials depending on the service you're

looking for. The Junior's section of the website does display bank transfer details, but a prospective member must email Russell to confirm payment. For both forms of membership, payment must be done manually every season, or if paying in parts; "four equal monthly instalments at the start of May, June, July and August." (Welwyn Garden City Cricket Club, n.d.)

2.2 Programming Languages

Choosing a suitable programming language based on my current skills and what I want to build is an important theme in my project. The language is not necessarily the deciding factor for the outcome of the project but will certainly affect how I approach the problem. There are many languages that could be used to build a website such as JavaScript, Python, PHP, Ruby, and HTML/CSS.

A website makes use of both front-end and back-end technologies. The front-end is where the user can interact with the website though a Graphical User Interface (GUI) where they can navigate through pages and visualise images, videos, and text. Whereas the back-end contains code that the user cannot see and determines how the website works. Examples of front-end languages include HTML/CSS and JavaScript. Back-end examples consist of PHP, Ruby, Python and, again, JavaScript, which can be used as both. (GeeksforGeeks, 2021)

2.3 Data Management

Storing data is a critical theme I wanted to explore as part of my research. Data makes the current world turn and is used, in essence, as a commodity in today's digital age. It acts as an asset to any business, not necessarily to sell, but to be able to process into usable information to boost business and increase customer satisfaction or sell at a high value. (A. Moka, 2020). Keeping that in mind, storing data securely and efficiently is incredibly important regardless of how large a business is, whether it's small-scale like my project or a big data corporation like Meta (formally Facebook).

A Database Management System is an important tool when developing web applications or websites. It allows you to effectively store all sorts of data and then interface the data through read & write protocols. A Relational Database Management System (RDBMS) would be ideal for my artefact as the data is categorized into definitive tables and would fit the scale of operation.

"Together, ACID is a set of guiding principles that ensure database transactions are processed reliably. A database transaction is any operation performed within a database, such as creating a new record or updating data within one." (Watts, 2020). Making use of the ACID reliability model can help ensure the Atomicity, Consistency, Isolation and Durability of a database. A database management system such as MySQL uses the InnoDB storage engine which maintains the ACID model to protect user data when carrying out transactions of data.

Abiding by GDPR is important when it comes to storing data on users. Keeping information up-to-date and relevant is a key factor, along with restricting access to data to only the people needed it. People also have the right to erasure if they feel their data is no longer needed. In terms of data held on children, "only children aged 13 years and over may lawfully provide their own consent for the processing of their personal data;" whereas "an adult with parental responsibility must provide consent for processing if the child is under 13" (ICO, n.d.)

2.4 Website Security

When a user enters a website, they want to ensure that what they have clicked on is a genuine web-site/page. Hackers will always to try to find weaknesses in a website and there are many forms of attack that could be carried out. “Ecommerce sites will always be a hot target for cyberattacks. For would-be thieves, they are treasure troves of personal and financial data.“ (Big Commerce, n.d.). E-Commerce sites are obviously more at risk due to their involvement with online payments and storage of highly confidential user data, therefore require much greater vigilance when making sure your website is secure.

There are many factors that can determine the authenticity of a website to help give yourself and customers that peace-of-mind. According to Drew Hendricks (n.d.) some of these include:

- Use of HTTPS
- Secure Web-Hosting
- Keeping database and root/user passwords secure and strong

HTTPS is an acronym for ‘Hyper Text Transfer Protocol Secure’. This type of protocol utilises Secure Socket Layer (SSL) or Transport Layer Security (TLS). It provides encryption of any information passed from the client/user to the server. “ TLS is an authentication and security protocol widely implemented in browsers and Web servers. SSL works by using a public key to encrypt data transferred over the SSL connection.” (HealthIT.gov, 2019)

Having secure web-hosting is an important factor in keeping your website secure. If you are out-sourcing your hosting to another company, they should be providing a secure service. However, your website will only ever be as secure as you make it. If there are vulnerabilities in your website, the responsibility would ultimately fall on you. Web-hosting services, according to digital.com (2021), should make sure of physical hardware security, network monitoring, provision of secure access over a protocol such as Secure Socket Shell (SSH) which uses password and public key authentication and, should make back-ups to ensure disaster recovery.

Finally, having secure passwords is basic security. As a website administrator you do not want unauthorised persons accessing your website’s database(s) or back-end services as such a thing could be disastrous to the business. According to IBM (2021), 2021 has been the worst year in 17 years in terms of data breach costs and the most common attack was passwords that were compromised; contributing to 20% of breaches. In the context of protecting user data, password cryptography, in the form of hashing, is a popular form of security used by most websites. Common hashing algorithms include SHA-1, SHA-2 and MD5. According to Oriyano (2016, p87), MD5 is being phased out and replaced by more secure crypto-hashes such as SHA-2 due to it officially being compromised and is something that should no longer be used.

2.5 Payment Gateway Providers

“A payment gateway solution offers a host of benefits to users. It enables you to sell online by allowing you to charge the purchase amount to your customer’s credit or debit card“ (Baker, n.d.)

Payment gateways are a useful tool when creating an e-Commerce feature of a website, especially in today’s modern world. As shown above, it enables people to pay online. As I’ve researched, most payment providers offer Application Programming Interfaces (API). An API is like a gateway that enables a business to safely open their data and functionality to third-party

developers. This creates a free or paid service from the business and an easy-to-use interface for developers. (IBM, 2020) Using a Payment Gateway that provides API would allow me to implement online payment processing into my solution. Multiple Payment Processors provide Gateways with APIs, but many require monthly subscriptions on top of transaction fees. Obviously, the Payment Processor is offering a service, such as PayPal or Lloyds Bank.

For the sake of my artefact, I didn't want to use a Payment Processor that would charge a monthly fee for use of their Payment Gateway, so instead opted for ones that only charged a small fee on transactions. Ross Darragh at Startups.co.uk on the 8th of November 2021, created a collation of various Payment Gateways and their prices which gave me a starting point, of which here are a few:

- Worldpay - £19/month
- PayPal - 2.9% + 30p/transaction
- Stripe - 2.9% + 20p/transaction
- Amazon Pay – 2.7% + 30p/transaction

2.6 Push Notifications

Push notifications are an incredibly effective way of communicating with users and allows the quick transfer of information from the business to the consumer. There are various types of push notifications that can be utilised, and each have their advantages, disadvantages, and suitability for certain types of information. Some are relevant to the scope of my artefact such as Emails, SMS, and Desktop Push Notifications.

"SMS is effective if you want your user to read your message very quickly – 90% of SMS messages are read within the first 3 minutes." (Kansal, n.d.). SMS messages are incredibly vital for relaying information that doesn't require much thinking and serves as a form of reminder, urgent news, or any kind of information that doesn't necessarily require an instant response. According to Anand Kansal, SMS messaging is not suitable for sending receipts or large pieces of information that may need future access. This is because of character-limits and difficulty in re-referencing when needed.

This disadvantage, however, is where email notifications can come in handy. Email notifications are ideal for digitalising important information such as invoices or receipts. It also serves as a way for businesses to offer marketing options to customers, this however would not be relevant to the scope of my artefact.

2.7 Review Summary

Information and literature on web design and structure is incredibly plentiful and this review was made to outline a few that will be relevant to the scope of my artefact. It's allowed me to research and explore ways in which to implement my solution with the vision I have currently.

As my artefact deals with a financial solution, security is a very important theme. A lot of my themes tie in together in that regard. Exploring a suitable programming language and discussing security issues & data management are all topics that ensure a website to be usable in the sense that customers or members would feel safe inputting their information and financial data. Security from a language standpoint is an important cornerstone and some make encrypting data far easier than others with good documentation on how to do so. Discovering a suitable database to store this encrypted data should utilise strong and secure password access.

Online payments have become widely popular and many financial institutions such as Lloyds Bank and online financial businesses, like PayPal have started creating their own payment gateways for online e-commerce opening a whole new stream of revenue to them and offering a service that increases efficiency of payment and customer satisfaction in a rapidly evolving digital world.

3.0 Project Plan

My Project Plan will determine a detailed plan of my software implementation, discuss risks associated with pre-implementation and post-implementation and uncover stakeholders who may be affected by my solution. It will also utilise a Gantt Chart found in **Appendix A (p. 53)**.

3.1 Proposed Solution

After analysis of the current solution, it's clear to me that it does not offer an automated payment service and is completed manually by specific Club Officials. The idea is to take a web-based approach, because WGCCC already has a webserver with a website and to create push-notifications alongside it. For the artefact I've decided to take a MoSCoW approach, to include 'must haves', 'should haves', 'could haves' and, 'would haves' for the requirements. It's clear that re-coding the entire website would likely fall outside time constraints and would come under a 'could have'. Push-notifications would be a 'should have' and the payment system is a 'must have'.

Per my findings and research in my literature review, I discovered multiple pathways in which this artefact could be implemented. Whether that be a certain language, various technologies, and payment system approaches.

The plan isn't to create a whole new payment processing system as it would ultimately take far too much time. Instead, I researched various providers that offer different services for different price plans. It ranged from transactions fees to up-front costs. Finally, I decided that I'm going to go with Stripe as my desired Payment Gateway Provider. This is because of the cheaper prices at just 20p per transaction and their test suite which means real money doesn't need to be used in development mode. The percentage-take difference between the providers was not too much of a deciding factor. However, I also chose to go with Stripe because of their clear and concise Python API Documentation.

Creating a web-application will require both a front-end and a back-end. It's important to finalise what languages I want to use to develop my artefact. As hinted above, I will be using Python as my back-end language due to my already basic experience with it. It's still a learning curve, but it isn't completely alien to me. There's plenty of available documentation for it. On top of that, I decided that if I'm going to use Python, I should use HTML and CSS as my front-end languages. JavaScript may also make an appearance, but due to my current knowledge of JavaScript, it's very likely I'd be learning it as I go or beforehand.

Web applications such as ones that provide payment processing, require security because of the nature in which a user is using their sensitive payment data. I decided that using a relational database would be optimal. Python offers packages that work brilliantly alongside a database, especially because of Python's many possibilities of modularisation, I can add a password hashing module. It would allow me to store data securely. The database management system I decided to use is MariaDB as it's a system that is currently already available to me and is incredibly similar to MySQL. This isn't because it's any better than other database management

systems such as Oracle, but because I already have basic prior usage with it and I far much prefer the MariaDB/MySQL syntax.

Push-notification services will also utilise a paid API as an overhead to the business. Costs come in the form of sending SMS texts to members. I decided to use TextMagic as it offers simple implementation and a Python wrapper module.

3.2 Preparations

3.2.1 Support & Development Tools

GitHub Desktop

GitHub Desktop will allow me to create a local repository and regularly backup commits to a repository on my account at https://github.com/. Access to this repository will be offered to my project supervisor. I have used this previously in personal and university projects when undergoing individual and/or paired projects as a robust versioning system and have good knowledge how to set up local repositories.

Atom IDE

This Integrated Development Environment interfaces well with GitHub Desktop for easy committing. It also offers many useful packages that can be downloaded. For example, I can create hotkeys to automatically run Python files without having to navigate to the directory.

Flask Development Webserver

Python Flask has its own in-built server that runs on Port 5000 by default. It'll allow me to run my app on Localhost for development purposes.

HeidiSQL

This database administration client will allow me to interface my database and make manual changes & additions.

3.2.2 Functionality Tools

MariaDB RDBMS

This relational database management system will be relevant for my project due to its scale and what I want to store. I have used MariaDB before which is why I decided to choose it. I prefer the syntax over most other RDBMSs I've used, such as Oracle.

Python

This language will be used for the back-end. Effective for use in web applications with plenty of documentation and support material. I've used this before but have only touched the basics of web development with it. It will certainly be a challenge to create a working artefact with this. It contains dependencies that can interface MariaDB such as 'PyMySQL'.

HTML/CSS

A vital front-end pair that works effectively alongside Python. These have also been around for a long time and offer plentiful support material and documentation. Used in web development for an incredibly long time. My experience with it is also limited and I have also only ever touched the basics.

Python Flask

Flask Framework, especially made for Python, allows for web implementation using Python and HTML/CSS/JavaScript. As mentioned in the Development Tools, it also provides the Development Webserver.

Jinja Template Engine

Jinja allows me to pass Python variables and code to the front-end of my website. An incredibly useful tool that opens a lot more possibilities of implementation. The template engine comes with Python Flask by default

JavaScript

JavaScript is a useful language for creating dynamic pages and functions on a webpage. It, too, works well in web development alongside HTML. There is also a plethora of documentation on it. My experience with JavaScript in web applications is limited as I have only ever used it with Node JS alongside a Discord JS framework to create bots. However, this was heavily modified JavaScript. I do plan on improving my knowledge of this language before undertaking my project.

Stripe API

The Stripe API is what I'll be using for my payment processing gateway. It's a paid service but, offers a test-suite so I don't need to pay anything. It provides Python Documentation which is incredibly useful because I have never used this before. Implementation of Stripe will be completely new to me.

TextMagic API

TextMagic is what I'll be using to setup SMS Notifications. It is also a paid service, charging per SMS sent and received. I have also never used this API before, and implementation will be a new experience.

3.3 Stakeholders

This list of stakeholders will outline who will be affected by this project

- Club Officials
    - This is a managerial role of the Cricket Club
- Club Members
    - These are members who have paid for membership, other subscriptions, or any kind of paid services from the club.

- Prospective Members
  - These are possible members who may or may not choose to opt for a paid service
- TextMagic
  - If the SMS Notification service is a success, this provider will gain monetary value from this project when SMS notifications are sent to Club Members or Prospective Members.
- Stripe
  - Once payment processing is setup, this provider will also gain monetary value from this project per successful transaction.
- Project Markers
  - The two people marking this will assess the project. This includes my project supervisor and second marker.
- Me
  - I, of course, am a stakeholder in this project as I will be undertaking its development and implementation.
- Course peers
  - Throughout development, I may approach my peers in search of advice or opinions
- Employers
  - My project source-code may be able to be used in a portfolio for future employers to gain a grasp on my abilities when I enter the jobs market.

3.4 Risk Assessment

Predicting, preparing for, and managing risks is vital to ensuring the project runs as smoothly as possible. Below is a series of tables outlining possible risks and their methods of mitigation should they arise.

3.4.1 During Implementation

Table 1: A table to show risks and mitigation during the implementation of the project

Risk	Mitigation
CRITICAL: My hard drive corrupts, and I lose my Project. **Likelihood:** *Unlikely* **Consequence:** Loss of progress.	Back-up regularly to a cloud service – in my case: GitHub. Back-up regularly to another local drive.
I lose track of time or fall behind for various reasons including, but not limited to 'Scope creep'. **Likelihood:** *Likely* **Consequence:** Project won't be fully completed by the deadline.	Using a service like Trello to keep track of modules that need completing on the Project and their progress. Making use of a General-use Gantt Chart.
A main stakeholder, Club Official, offering the requirements, changes their mind on a specification **Likelihood:** *Unlikely* **Consequence:** I'll need to change my project plan depending on the scale of change.	Review the changes and discuss with the Club Official the likelihood of implementation and adapt should it be agreed – depends on timeframe, my experience, and confidence.
CRITICAL: I produce bad-quality code that isn't modular, readable, or commented effectively **Likelihood:** *Unlikely*	Ask my peers to review my code regularly. Regularly review my own code and comment as I go.

Consequence: Reviewing my own code will become almost impossible should I need to go back to it.	Commit to some coding standards.
CRITICAL: Poor Risk Management **Likelihood:** *Unlikely* **Consequence:** Issues may arise that I'm not prepared for and haven't thought about.	Complete a risk assessment, exactly like this one, and acknowledge that risks can in fact occur.
Lack of knowledge in some areas **Likelihood:** *Likely* **Consequence:** The project will not be how I envisioned originally.	Do some pre-reading of features/technologies I'm unfamiliar with.

3.4.2 Post-Implementation

Table 2: A table to show risks and mitigation after the implementation of the project

Risk	Mitigation
The software encounters some bugs on live production **Likelihood:** Likely **Consequence:** Depending on the scale and criticality of the bug, the software will not work as intended	Work with the given bug report or finding and aim to patch it as soon as possible depending on criticality.
CRITICAL: Club Officials have no idea how to use the new system **Likelihood:** Unlikely **Consequence:** Creates a disconnect between me and the client. Club Officials are supposed to know how things at the club work.	Document all features in a user-friendly way.
CRITICAL: Management completely change their mind about the software and no longer want to use it **Likelihood:** Unlikely **Consequence:** No further software integration.	Discuss with the Club Officials about their justification to see if improvements to the software can be made. If not, respect their wishes.
CRITICAL: Some code from the APIs used in implementation become deprecated. **Likelihood:** Unlikely **Consequence:** The software no longer works as intended.	Ensure maintainability of software – whether yourself or someone else. Maintain and update the deprecation.
CRITICAL: A bug on a third-party module is found and is out of my control to find a fix for it. **Likelihood:** Unlikely **Consequence:** The software no longer works as intended or at all.	Remove the third-party module and find an alternative third-party service.

3.5 Statement of Success

Besides creating an artefact, it's important that I learn from my experiences and determine my success of my academic achievement throughout the project rather than solely how successful the implementation of the artefact turns out. The basis of the project is to put me in a "real-world" scenario, assesses how I would approach such an issue, and what I can learn from it.

Through the duration of the project, I hope to gain valuable time-management skills tailored to the Software Development process. This is an aspect of development that will be vital for future endeavours when I graduate and am looking to kickstart my career in the Software Engineering/Development industry. It's a skill that employers will expect of me, and this project will help me self-assess my ability to do so and improve where necessary.

Another skill I hope to enhance is my ability to adapt to a changing situation. It is hoped that the requirements don't need to change, but if they do for whatever reason, such as a feature not working as I had hoped, then I will need to learn how to overcome the situation and find an alternative effectively. I should be prepared for that.

The importance of this project towards my final grade has put me in a position where I must focus and complete the project to the best of my ability. I have never taken an approach to personal, impromptu, projects too seriously, and I feel this will be a great opportunity to learn how to dedicate time and effort into something of such importance, a mind-set which may also reflect future projects when it comes to it.

4.0 Design

4.1 User Stories

Table 3: A table of user stories for functional requirements

- User Stories - Functional Requirements		
As a…	**I want…**	**So that…**
Website user	to login	I can access available subscriptions
Website user	to view available subscriptions	I can purchase one
Club Official	to access existing customers	I can monitor activity
Website user	to access my profile	I can view and edit my information
Existing Customer	to receive a push notification	I can keep track of invoices
Club Official	to manage available subscriptions	I can keep products up to date
Existing Customer	to access my profile	I can easily view active and past subscriptions
Website user	to input and hold details on a junior	I can purchase a subscription for a junior

Table 4: A table of user stories for non-functional requirements

- User Stories - Non-Functional Requirements		
As a…	**I want…**	**So that…**
Website user	a clear sitemap	I can navigate the website easily
Website user	the webpage to have a suitable colour-scheme	It reflects the purpose of the website
Website user	Fast loading speeds	I can have a smooth experience

4.2 Storyboards

The storyboards/wireframes I plan to showcase in this section of the report have taken inspiration from the original website. Users who use the original site may not be used to a big change, so I wanted to keep it similar. A study conducted by a PhD Informatics student named Sam Goree from Indiana University stated that similar websites and libraries are "more user-friendly, since new visitors won't have to spend as much time learning how to navigate the site's pages"

Home Page

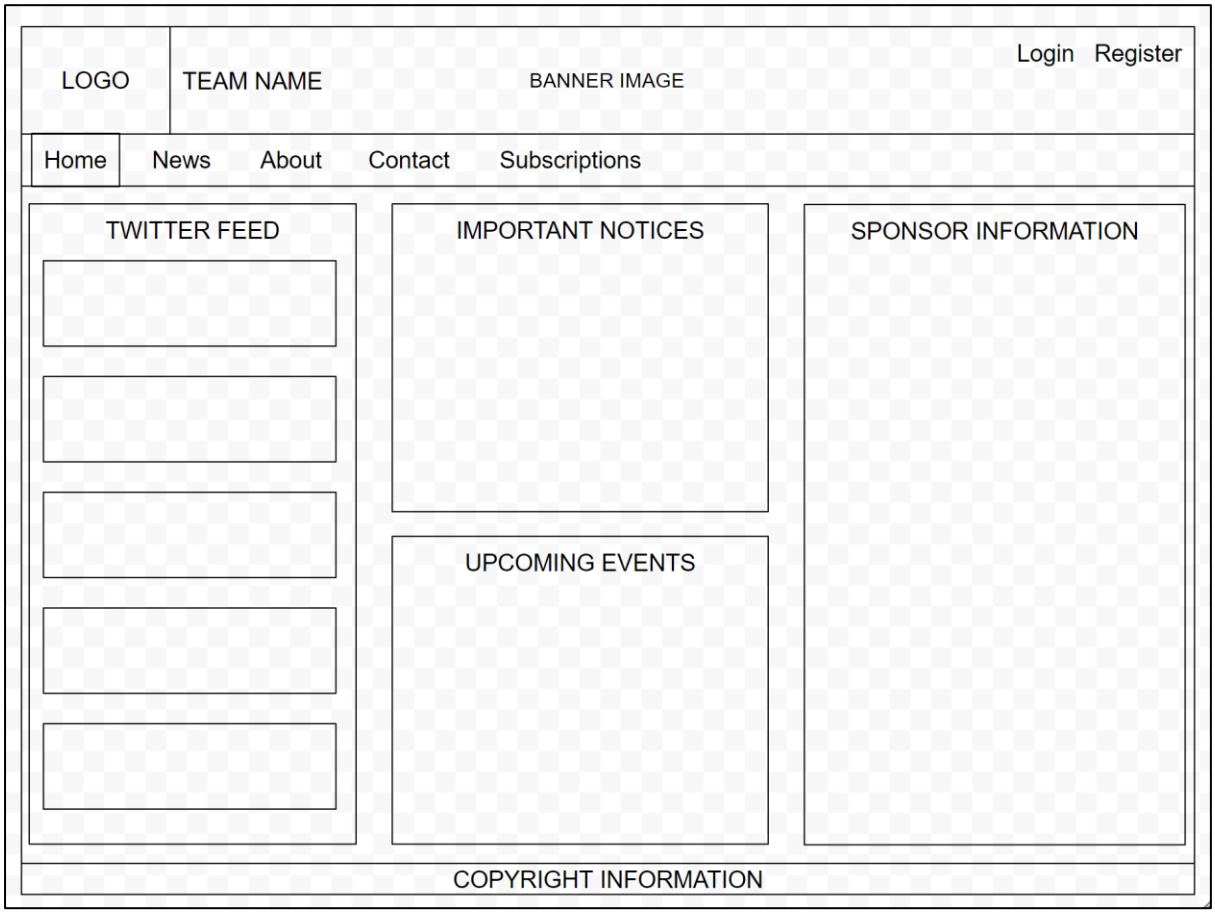


Figure 1: Home screen storyboard

As shown in the storyboard in ***Figure 1***, the main template of the website would display the club's logo and name in the banner. The banner image would likely reflect the theme of Cricket. The navigation bar is simple and defines, explicitly, the sitemap. This makes it simple to traverse the website. In the top right corner, we have access to login or register. I've decided to place it at the top so users can see it easily on opening the page. At the bottom of the page, we have copyright information. These aspects will be global across all pages.

Unique to the home page, it will sport a twitter feed which would contain regular updates and history of tweets from @WelwynBees. It would be vertically scrollable. Important notices from Club Officials will also be placed on this page. Importantly, sponsor information is placed on the right. Upcoming events will also be displayed. This could display upcoming matches or club events such as charity events.

Register Page

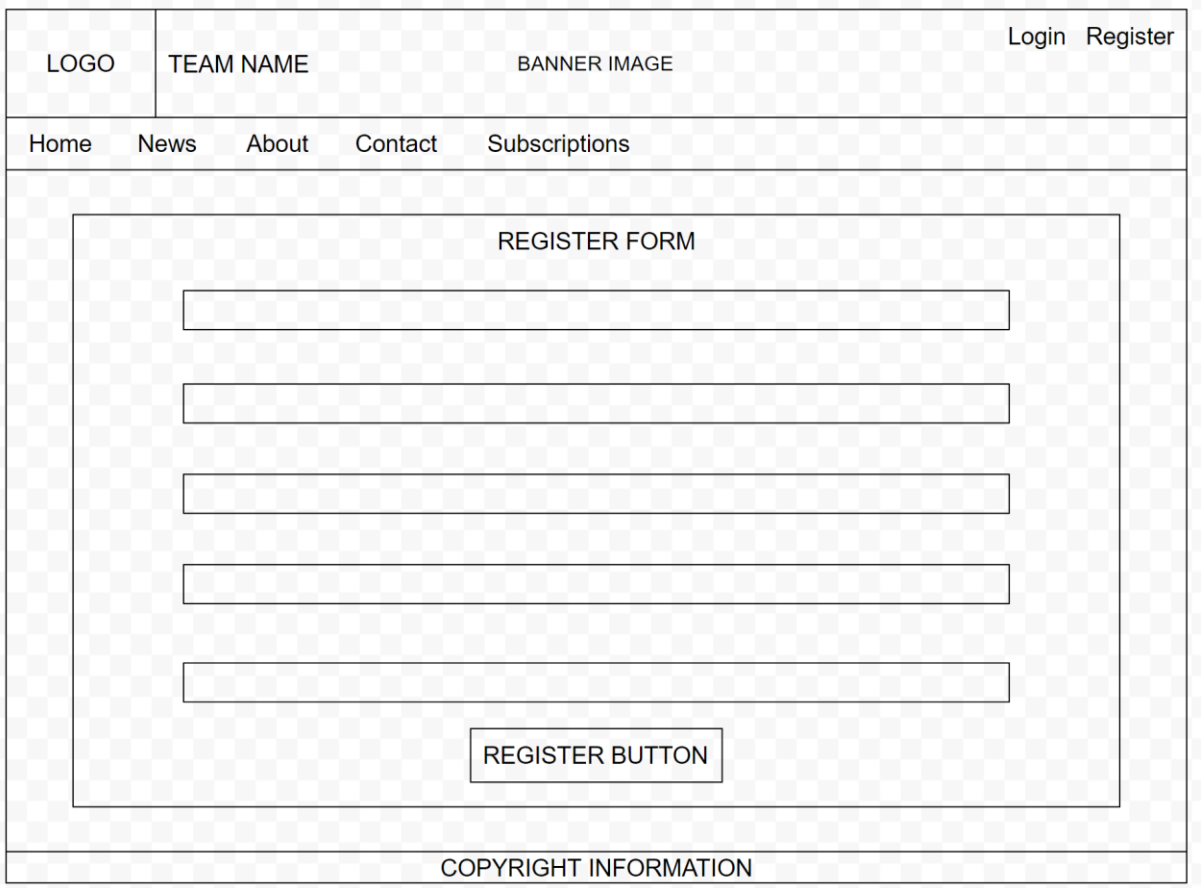


Figure 2: Register screen storyboard

The registration page will be a simple form that will collect information about the prospective user. The user will fill in text inputs. Data that is collected will be discussed further, in the database structure section of this report.

Once the user has input their information, they can use the 'Register Button' to submit their details to the server.

Login Page

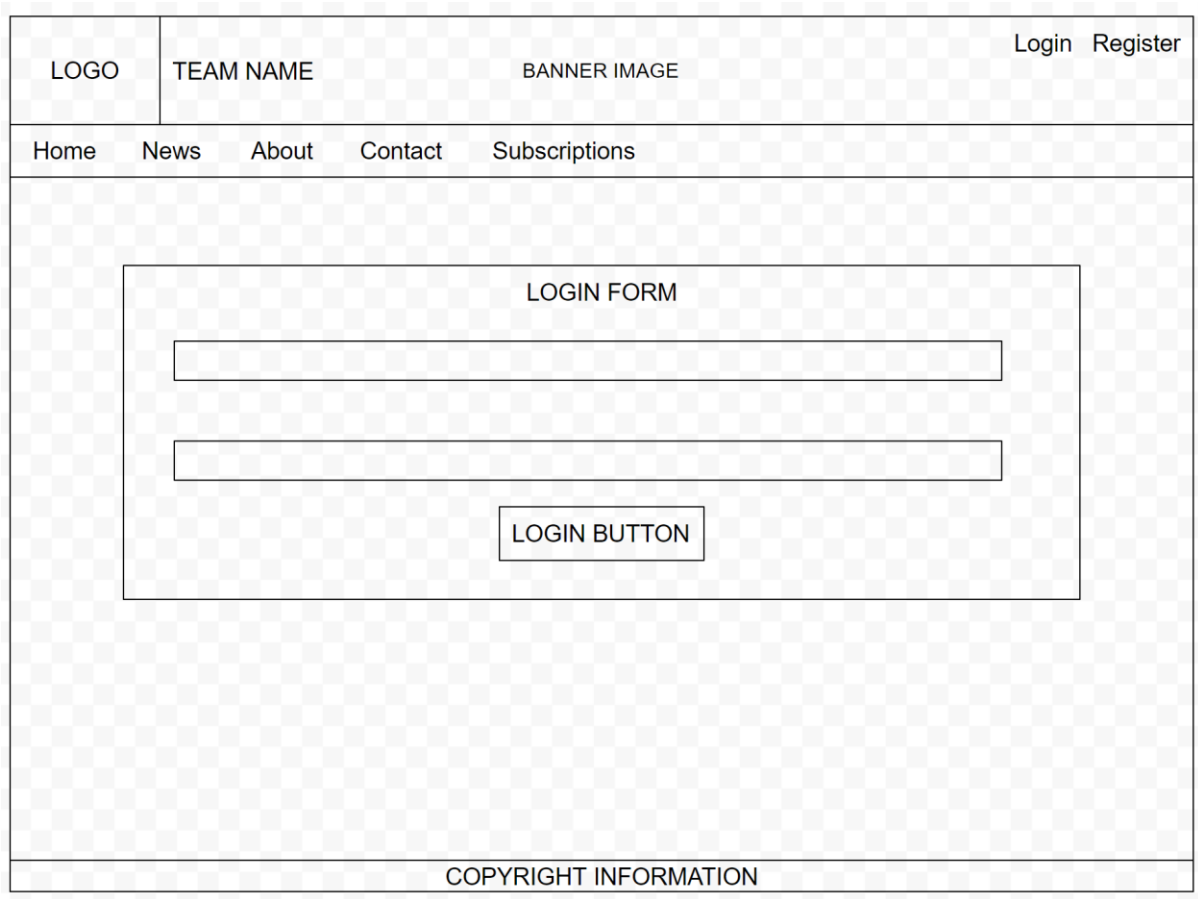


Figure 3: Login screen storyboard

Like the Register page, the Login page will also be a simple text input. It's extremely likely that it will just collect the user's email and password.

Subscriptions Page

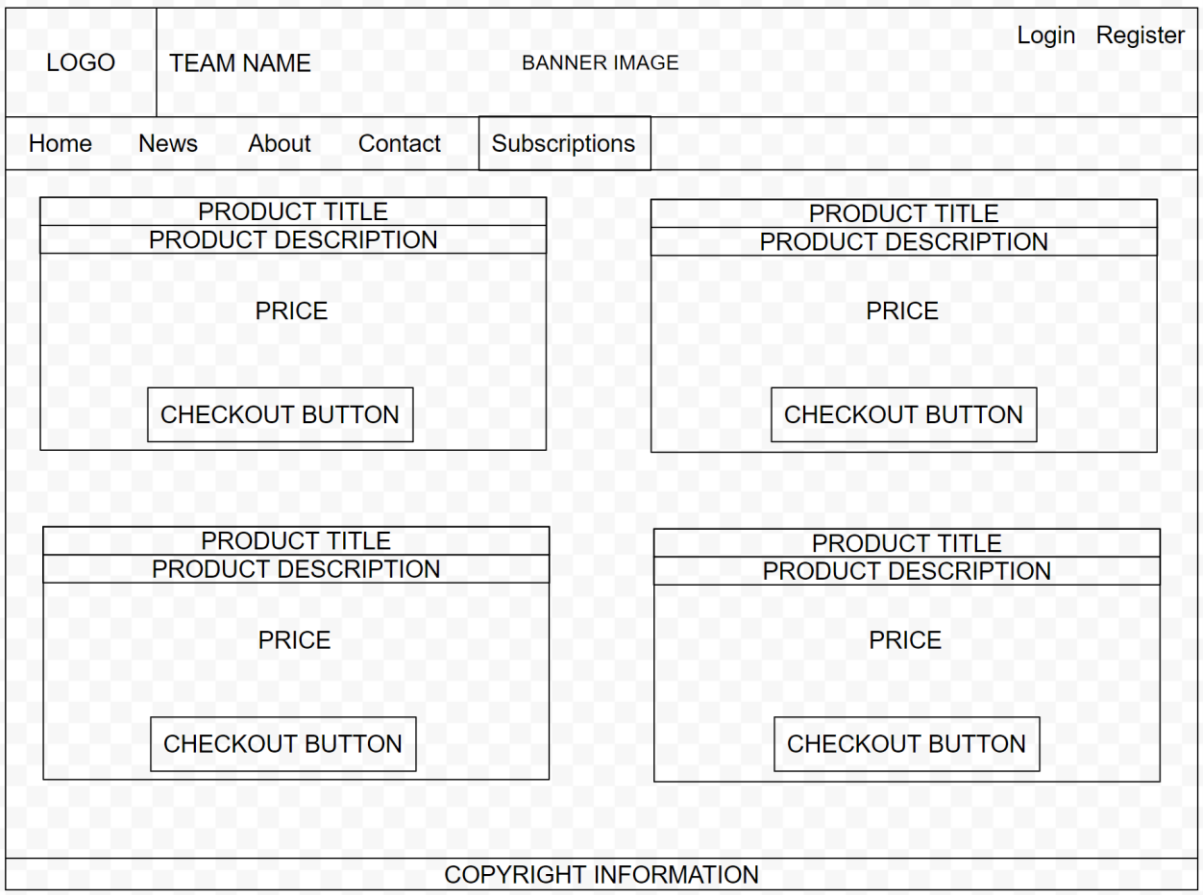


Figure 4: Subscriptions screen storyboard

The subscriptions page is the most important part of this project. It will be the main design of the artefact. It must be clear and concise showing what product is what. Having products in their own individual containers with their own respective information is vital to ensuring clarity. Each product container will have their own product name, description, and price. The containers will contrast against the page background. In the case of overflow, such as there being more than four products, the page will become vertically scrollable. A search bar could also be a possibility, but due to the size of operation, it doesn't seem necessary at the moment.

Upon the checkout button being pressed, the user will be taken to a third-party page to proceed with payment.

Profile Page

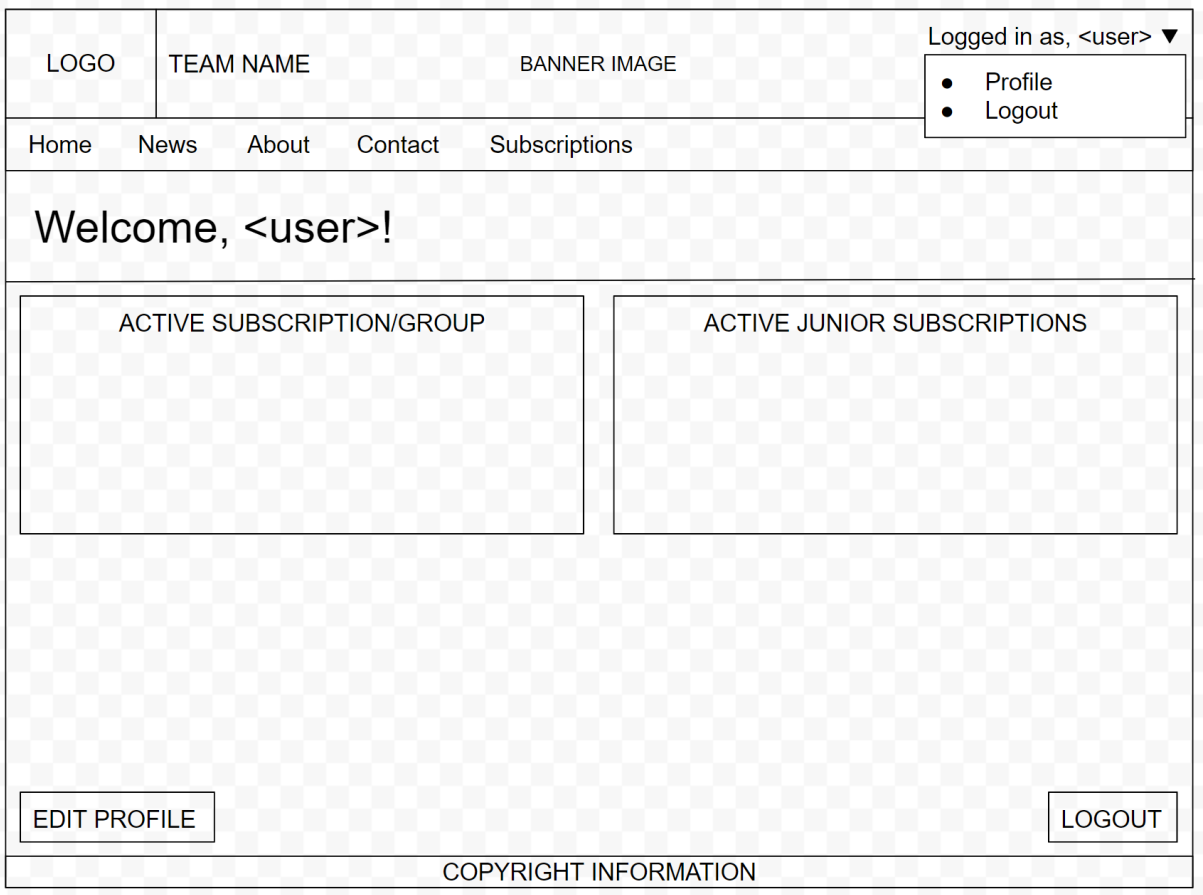


Figure 5: Profile screen storyboard

When a user logs in they will be able to access their profile through a drop-down menu on the top right of the screen where they see their name. This is not something unique to the profile page but will be global.

Upon entry of the profile page, the user will see a greeting and some containers. One of these containers will contain the active group the user is in or the subscription they currently have active, if any. They may be able to edit their subscriptions from here, but for the time being there is no plan to implement this. Apart from this, if a user has Junior subscriptions active on their account, they will see details of their child(ren) and the active subscription. For child safety reasons, this information will only ever be minimal.

4.3 Database Structure

4.3.1 Entity-Relationship Model

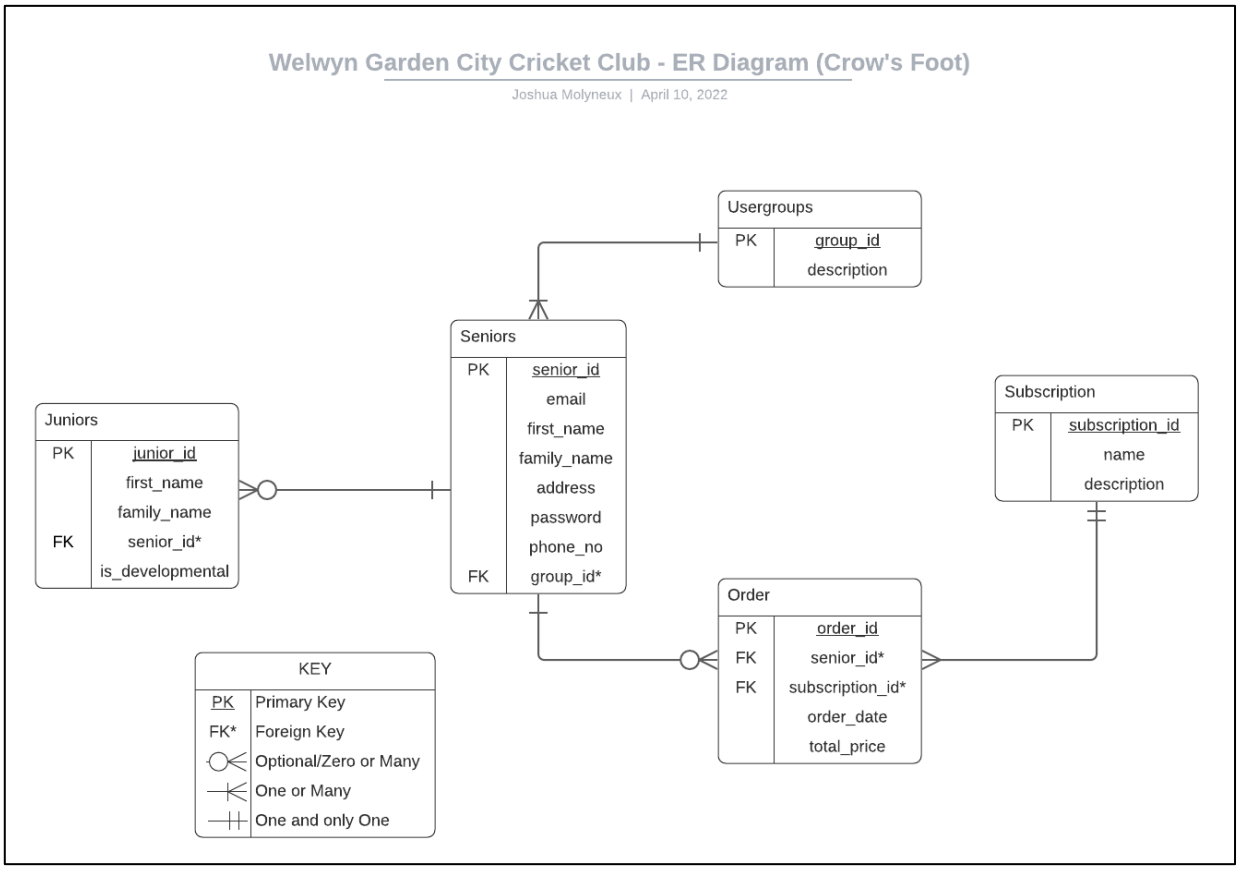


Figure 6: Crow's Foot Entity-Relationship Diagram displaying the database structure

The structure of the database, in its current form, consists of five tables. User information will be stored in the 'Seniors' table. Information collected will include an identification number, the user's email, their name, address, password, and phone number. A foreign key stemming from the 'Usergroups' table will also be assigned to the user.

The 'Juniors' table will also assign an ID number. Their full name is also stored in case the junior does not have the same surname as the senior member which could be a possibility, so I must cater for that. A foreign key from the 'Seniors' table will tell me who the junior belongs to. The final field, 'is_developmental' will determine what subscription the junior has (whether under 7/8 years old). A senior can have zero or many juniors connected to their account.

The 'Usergroups' table will be used for profile purposes. It's simply something that could be used in the implementation stage to decide what users can see what. Plans are to have usergroups for the certain subscription a user has, an admin usergroup, or a default one for all users on the website. A simple description will accommodate it.

The 'Order' table will be used to collect information about subscriptions bought by users. It utilises multiple foreign keys, one to determine which senior bought something, and the other to determine which subscription it is that was purchased. Date of purchase and its price is also stored.

As aforementioned, the 'Subscriptions' table will be a reference table containing static information about available subscriptions.

4.3.2 Physical Model (MariaDB Syntax)

Seniors(senior_id, email, first_name, family_name, address, password, phone_no, group_id*)
Juniors(junior_id, first_name, family_name, senior_id*, is_developmental)
Usergroups(group_id, description)
Order(order_id, senior_id*, subscription_id*, order_date, total_price)
Subscription(subscription_id, name, description)

```
CREATE TABLE usergroups(
    group_id INT(6) PRIMARY KEY AUTO_INCREMENT,
    description VARCHAR(255) NOT NULL
);

CREATE TABLE seniors(
    senior_id INT(6) PRIMARY KEY AUTO_INCREMENT,
    first_name VARCHAR(40) NOT NULL,
    family_name VARCHAR(40) NOT NULL,
    email VARCHAR(100) NOT NULL UNIQUE,
    password VARCHAR(100) NOT NULL,
    address VARCHAR(255) NOT NULL,
    phone_number VARCHAR(30),
    group_id INT(6),
    FOREIGN KEY (group_id) REFERENCES usergroups(group_id)
);

CREATE TABLE juniors(
    junior_id INT(6) PRIMARY KEY AUTO_INCREMENT,
    first_name VARCHAR(40) NOT NULL,
    family_name VARCHAR(40) NOT NULL,
    senior_id INT(6),
    is_developmental BIT(1) DEFAULT 0,
    FOREIGN KEY (senior_id) REFERENCES seniors(senior_id)
);

CREATE TABLE subscriptions(
    subscription_id INT(6) PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(40) NOT NULL,
    description VARCHAR(255)
);

CREATE TABLE orders(
    order_id INT(6) PRIMARY KEY AUTO_INCREMENT,
    senior_id INT(6),
    subscription_id INT(6),
    order_date DATETIME NOT NULL,
    total_price DECIMAL(5, 2) NOT NULL,
    FOREIGN KEY (senior_id) REFERENCES seniors(senior_id),
    FOREIGN KEY (subscription_id) REFERENCES subscriptions(subscription_id)
);
```

5.0 Implementation

Setup and configuration is the first step in creating my artefact. I first need to create the project folder, which will be called 'www', some Python files, and an .env file for my environment variables. Using __init__.py, I can create functions that will load on initialisation. In the initialisation file, I first want import Flask and other dependencies and then grab the .env file as shown below in **Figure 7**.

```
from flask import Flask, redirect, url_for, session, flash
from dotenv import load_dotenv
import os

project_folder = os.path.expanduser('www')  # adjust as appropriate
load_dotenv(os.path.join(project_folder, '.env'))
```

Figure 7: Importing initial dependencies and getting .env file

5.1 Flask

From this, we can start creating our 'create_app' function. This will be the function that creates a Flask instance. See **Figure 8** below. According to Flask Documentation, it's common to use __name__ as the parameter. "The idea of the first parameter is to give Flask an idea of what belongs to your application. This name is used to find resources on the filesystem, can be used by extensions to improve debugging information and a lot more... If you are using a single module, __name__ is always the correct value." (Flask, n.d.)

The application requires a randomised secret key to function, so for security purposes I have placed it in the .env file and assigned it to the app config. After this, I can begin to create some blueprints, which are objects that can help me modularise my application. 'Views will refer to my main pages and 'auth' will point to my authentication pages such as the login or registration. When registering the blueprints, I can define how they can be accessed. I have no special requirement to do so, so I'll just keep it as default.

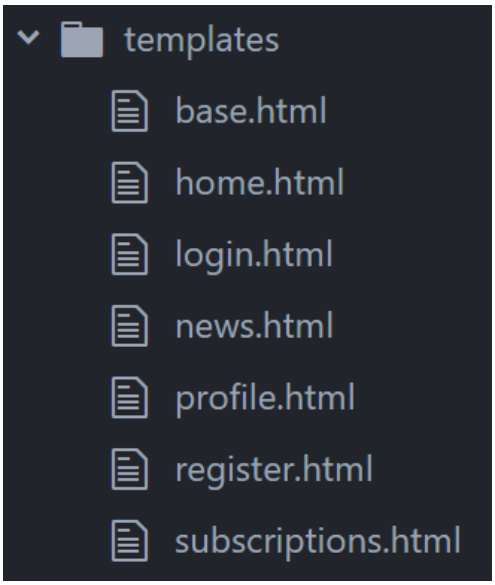
```
def create_app():
    app = Flask(__name__)
    # The app needs a secret key to work, grab it from the .env file
    app.config['SECRET_KEY'] = os.getenv("SECRET_KEY")

    # Create our page blueprints
    from.views import views
    from.auth import auth

    app.register_blueprint(views, url_prefix='/')
    app.register_blueprint(auth, url_prefix='/')

    return app
```

Figure 8: Creating a Flask application function

```python
from www import create_app


# Create a new application
app = create_app()

# Run our app
if __name__ == '__main__':
    app.run(debug=True)
```

Figure 9: Creating and running the app

In **Figure 9**, I can import my 'create_app' function to 'main.py' and create a new app instance. We also want to run it on the condition that the script is executed directly instead of being imported. For now, I want debug set to 'True' so that I can see any errors that may exist throughout development.



As shown in **Figure 10** to the left, I have created some new html templates that I will be using to create a front-end. 'views.py' and 'auth.py' will utilise these templates and render them. 'base.html' will be, as the name suggests, the base template of my application. Using the jinja template engine, I can make use of template extensions. For example, in 'home.html', I can include the 'base.html' and it'll show the base content along with its own unique content.

Figure 10: Directory structure of html templates

```python
from flask import Blueprint, render_template


views = Blueprint('views', __name__)


@views.route('/')
def home():
    return render_template("home.html")


@views.route('/subscriptions', methods=['GET', 'POST'])
def subscriptions():
    return render_template("subscriptions.html")


@views.route('/profile')
def profile():
    return render_template("profile.html")


@views.route('/news')
def news():
    return render_template("news.html")
```

Figure 11: Creating and rendering routes in 'views.py'

Starting with 'views.py', in **Figure 11**, I've created some placeholders for routes and the html pages they will render. After further research, I found that that routes represent the URL path and can utilise HTTP methods (TechWithTim, n.d.). For example, if a user goes to '/subscriptions', it will load and render the 'subscriptions.html' file. Routes allow me to specify what methods the page will use. They all use 'GET' requests by default, but I know for sure that subscriptions will use POST request information, so I need to explicitly define it. The same can be done for 'auth.py' for the following html files: login, register.

5.2 Database

```
import pymysql


def sql_connect(host, port, user, password, database):
    connect = pymysql.connect(
        host=host,
        port=port,
        user=user,
        password=password,
        database=database
    )
    return connect
```

Figure 12: Creating a database function to improve readability

Implementing the database from our database design requires a dependency called pymysql. As shown in **Figure 12**, I've created a function that I can call anywhere in my program to connect to my database. The parameters: host, port, user, password, and database are all required. Using this, I can now implement my physical model from my design material.

```
from www import create_app, sql_connect
import os

app = create_app()


@app.before_first_request
def before_first_request_func():

    SQL_HOST = os.getenv("SQL_HOST")
    SQL_PORT = int(os.getenv("SQL_PORT"))
    SQL_USER = os.getenv("SQL_USER")
    SQL_PASSWORD = os.getenv("SQL_PASSWORD")
    SQL_DATABASE = ''

    # Database connection
    connect = sql_connect(
        SQL_HOST,
        SQL_PORT,
        SQL_USER,
        SQL_PASSWORD,
        SQL_DATABASE
    )
```

In **Figure 13**, I can import the 'sql_connect' function from __init__ and use a useful Flask decorator called 'before_first_request'. The code block under this decorator will run before the first request on the application, as the name suggests. Next, I want to grab all our database information from the .env file. I decided to assign the database as an empty string for now because I want to do some checks for database existence. We can now connect to the database.

Figure 13: Creating a database connection with environment variables

In **Figure 14**, I first re-assign the 'SQL_DATABASE' variable with the environment variable holding my database name. I then connect to the database with a cursor object (which allows me to interact with the database). As shown, I create the database if it doesn't exist. If it does exist, then it will ignore the query altogether. Commit the changes and close the connection. Now that we know for a fact the database exists, we can change the database parameters, replacing the empty string database name with our environment variable. In **Appendix B (p. 54),** you can find the queries which verify the existence of the database tables

```
SQL_DATABASE = os.getenv('SQL_DATABASE')
# Create a new database instance
cursor = connect.cursor()
# Create the database if it doesn't exist
cursor.execute(f"CREATE DATABASE IF NOT EXISTS {SQL_DATABASE}")
# Save changes
connect.commit()
cursor.close()

# Now we can grab the database we just created
connect.select_db(SQL_DATABASE)
```

Figure 14: Verification of database existence

5.3 Base Template HTML

As previously mentioned, the Base HTML template will be a global template. I plan to use Bootstrap to assist me in my design work. Let's add the scripts to the document header as shown in **Figure 15**. The title makes use of the Jinja template so we can edit it in other HTML templates.

```
<DOCTYPE html>
<html lang="en">
    <head>
        <meta
            charset="utf-8"
            name="viewport"
            content="width=device-width, initial-scale=1.0, maximum-scale=1.0, user-scalable=no" />
        <link
            rel='stylesheet'
            href='https://fonts.googleapis.com/css?family=Lato:300,400,700'
            type='text/css'>
        <link
            rel="stylesheet"
            href="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.css"
            integrity="sha384-Vkoo8x4CGsO3+Hhxv8T/Q5PaXtkKtu6ug5TOeNV6gBiFeWPGFN9MuhOf23Q9Ifjh"
            crossorigin="anonymous">
        <link
            rel="stylesheet"
            href="https://stackpath.bootstrapcdn.com/font-awesome/4.7.0/css/font-awesome.min.css"
            crossorigin="anonymous">
        <link
            rel="stylesheet"
            href="https://cdn.jsdelivr.net/npm/bootstrap-icons@1.3.0/font/bootstrap-icons.css">

        <title>{% block title %}WGCCC{% endblock %}</title>
        <link rel="icon" type="ico" href="./static/img/icon.ico">
    </head>
```

Figure 15: Adding Bootstrap to the HTML document header

```
<body>
    <section id="banner">
        <div class="banner-text">
            <a href="/">
                <img src=./static/img/logo.jpg alt="WGCCC Logo" class="logo">
            </a>
            Welwyn Garden City Cricket Club
        </div>
    </section>

    <nav class="navbar navbar-expand-lg navbar-dark bg-success">
        <button
        class="navbar-toggler"
        type="button"
        data-toggle="collapse"
        data-target="#navbar"
        ><span class="navbar-toggler-icon"></span>
        </button>

        <div class="collapse navbar-collapse" id="navbar">
                <div class="navbar-nav mr-auto">
                    <a class="nav-item nav-link" href="/">Home</a>
                    <a class="nav-item nav-link" href="#news">News</a>
                    <a class="nav-item nav-link" href="#about">About</a>
                    <a class="nav-item nav-link" href="#contact">Contact</a>
                    <a class="nav-item nav-link" href="/subscriptions">Subscriptions</a>
                </div>
                <div class="navbar-nav ml-auto">
                    <a class="nav-item nav-link" href="/login">Login</a>
                    <a class="nav-item nav-link" href="/register">Register</a>
                </div>
            </div>
        </nav>
```

Figure 16: Creating the banner and navigation bar

As seen above in **Figure 16**, I have split my html sections into separate modules. Starting with the banner, I've implemented the logo and club name as set out in my storyboard in **Figure 1**. When the logo is clicked, it redirects to the home page should the user want to return to the homepage quickly as an alternative to the navbar.

The navbar, with the help of Bootstrap, becomes collapsible upon page resizing to assist in mobile-friendly usage. The navigation items go from left to right and I've also decided that the Login and Register navigation buttons should go on the same line rather than in the top right as shown in the storyboards. I struggled to get them there. See the banner in **Figure 17**.

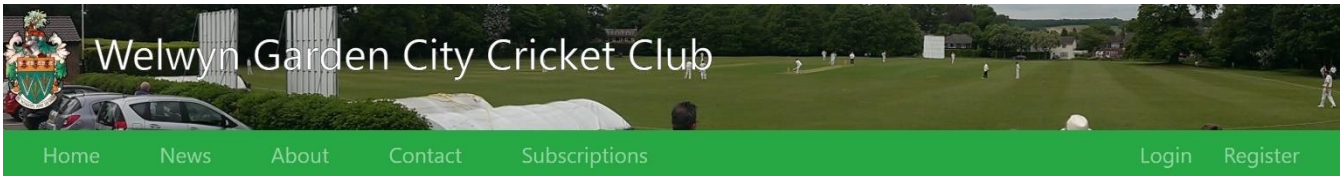


Figure 17: Website banner and navigation bar

The final two segments of the base template are the main container and the footer. See **Figure 18**. The main container makes use of the Jinja block utility. This will allow to me to extend my code and combine content from another page. The footer contains copyright information: my name and the year.

```
<!-- We'll place our custom content from oher pages here-->
<div class="container">{% block content %} {% endblock %}</div>

<section id="footer">
    <div id="copyright">
        <div data-inviewport="scale-in">
            <p class="text-light">Joshua Molyneux &copy; {{DATETIME}}
            </p>
        </div>
    </div>
</section>
```

Figure 18: Creating the main container and footer with copyright information

As seen next to my name, I've passed a Python variable to my front-end with the help of Jinja. I achieved this by amending the render_template in 'views.py' seen in **Figure 19.** I did this for all pages.

```
@views.route('/')
def home():
    return render_template("home.html", DATETIME=str(datetime.now().year))
```

Figure 19: Passing a Python variable to my front-end

Behind all this HTML lays CSS to support its styling. See **Appendix C (p.55)** for the stylesheet of the base template.

5.4 Registration

To start off the registration page, I have created a POST form in HTML to collect information from the user. See **Figure 20**. As you can see, I have started off by making the registration page an extension of 'base.html' and changing the title. I have then placed all the code in Jinja '{%block%}' tags.

```
{% extends "base.html" %} {% block title %}WGCCC - Register{%endblock%}
{%block content%}
<form method="POST">
    <h3 align="center"> Registration </h3>
    <div class="form-group">
        <div class="row">
            <div class="col-md-6">
                <label class="control-label" for="email">Email Address <span class='required'>*</span></label>
                <input
                    type="email"
                    class="form-control"
                    id="email"
                    name="email"
                    placeholder="Enter your email..."
                />
            </div>
```

Figure 20: Creating a POST form for the registration page

The data collected includes information found in the physical model of my database design. This POST form will comprise of the user's email, first name, family name, password, phone number, and address. To try and help stop human error, I thought it would be a good idea to research how Google's Place API could assist me in the use of auto-complete for the address fields. After some research online, I found that this was possible with the help of a user named David on StackOverflow (2015). I took inspiration from his answer and implemented it into my own work in JavaScript (**Appendix D (p.56)**). See my HTML code for the address form in **Figure 21.**

```
<div class="panel panel-primary">
    <div class="panel-heading">
        <h2 class="panel-title">Add your Address</h2>
    </div>
    <div class="panel-body">
        <input id="autocomplete" placeholder="Enter your address (auto-complete)" onFocus="geolocate()" type="text" class="form-control">
        <input type="hidden" class="field" id="street_number" disabled="true"></input>
        <input type="hidden" class="field" id="route" disabled="true"></input>
        <div id="address">
            <div class="row">
                <div class="col-md-6">
                    <label class="control-label">Address Line 1 <span class='required'>*</span></label>
                    <input class="form-control" name="address1" id="address1" disabled="false">
                </div>
                <div class="col-md-6">
                    <label class="control-label">County <span class='required'>*</span></label>
                    <input class="form-control" name="administrative_area_level_2" id="administrative_area_level_2" disabled="true">
                </div>
                <!-- <div class="col-md-6">
                    <label class="control-label">Address Line 2 </label>
                    <input class="form-control route" name="route" id="route" disabled="false">
                </div> -->
            </div>
            <div class="row">
                <div class="col-md-6">
                    <label class="control-label">City / Town <span class='required'>*</span></label>
                    <input class="form-control field" name="postal_town" id="postal_town" disabled="true">
                </div>
                <div class="col-md-6">
                    <label class="control-label">Postal Code <span class='required'>*</span></label>
                    <input class="form-control" name="postal_code" id="postal_code" disabled="true">
                </div>
            </div>
        </div>
    </div>
</div>
```

Figure 21: Address auto-complete HTML form

At the bottom of 'register.html', I have finished off the form by providing a submit button, including the API, 'auto-complete.js' file and closing off the {block} tag (**Figure 22**).

```
<button type="submit" class="btn btn-primary auth-submit">Submit</button>
<script type="text/javascript" src="./static/js/auto-complete.js"></script>
<script defer src="https://maps.googleapis.com/maps/api/js?key=AIzaSyCEm1-Vnk8OeHE_R4TT5-Vn8SW7iaVvpfQ&libraries=places&callback=initAutocomplete"></script>
{%endblock%}
```

Figure 22: Finalising the Registration page front-end

Now that I have completed the front-end design of the Registration page, I can look at implementing the back-end. In 'auth.py', I have begun to process the POST data sent to the server from the client by assigning them to variables in **Figure 23**.

```
# Retreive POST request content
if request.method == 'POST':
    email = str(request.form.get('email'))
    first_name = request.form.get('first_name')
    family_name = request.form.get('family_name')
    password = request.form.get('password')
    password_confirm = request.form.get('password_confirm')
    # Check if user input a phone number as it's not required
    if request.form.get('phone') is not None:
        phone_number = request.form.get('phone')
    else:
        phone_number = "NULL"
```

Figure 23: Assigning the POST data to variables

The address, however, will need to be handled differently if I know that I am going to be adding it to the database as single string instead of several. In reference to **Figure 24,** let's first create some logic that will help to compile this information. We can create a list of HTML Element names I assigned on the front-end along with an empty list with the values we attempt to collect. We also want to check if the address is complete, a Boolean flag can assist with this. We want to loop through the 'postal_keys' and get the request data for each element. If one of the elements contains an empty value, we can set the Boolean flag to False and break out of the loop and restart with some logic I will show further on. If the address values all come back with something, we can append them to our empty list and then finally join them together into one string separated by a comma stored in a single variable called 'full_address'.

```
postal_keys = [
    'address1',
    'postal_town',
    'administrative_area_level_2',
    'postal_code'
]
full_address_list = []
full_address_complete = True

for i in postal_keys:
    if request.form.get(i) is None:
        full_address_complete = False
        break
    else:
        full_address_list.append(str(request.form.get(i)))

if full_address_complete:
    full_address = ','.join(full_address_list)
```

Figure 24: Creating a full address string from various elements

It is important to validate the data sent from the client with some reasonable requirements. But we also want to provide the user some feedback as to what they got wrong (if they do). Flask offers a module called 'flash'. This creates a popup to the user with custom content. See **Figure 25**. After some research on Flask (n.d) Documentation, I found out how I could implement it into my 'base.html' using Jinja to pass some Python logic. As shown, we can define different flashes into different categories. With the help of Bootstrap we can style these separate categories. I have split them into success and error, but it is extendable should I need more in the future. The 'message' variable represents my custom messages. In Python, usage would look like: '*flash("There was an error", category='error')*'.

```
<!-- Create our Flash implementation for when we specify 'error' or 'success' in Python-->
{% with messages = get_flashed_messages(with_categories=true) %}
    {% if messages %}
        {% for category, message in messages %}
            {% if category == 'error'%}
                <div class="alert alert-danger alter-dismissable fade show" role="alert">
                    {{ message }}
                    <button type="button" class="close" data-dismiss="alert">
                        <span aria-hidden="true">&times;</span>
                    </button>
                </div>
            {% else %}
                <div class="alert alert-success alter-dismissable fade show" role="alert">
                    {{ message }}
                    <button type="button" class="close" data-dismiss="alert">
                        <span aria-hidden="true">&times;</span>
                    </button>
                </div>
            {% endif %}
        {% endfor %}
    {% endif %}
{% endwith %}
```

Figure 25: Implementing flash messages into 'base.html'

With that, we can begin the validation of input in **Figure 26**. Firstly, we want to verify whether the email a user is trying to register doesn't already exist. We can do this by creating a Boolean flag called 'email_exists' and assume it's False by default. Next, I have created a database connection that will execute a 'SELECT' query and compare our email variable with whatever result the query provides. If the query doesn't come with a result, we can assign our 'email_exists' flag with a True value and close the connection. Upon finding that the email exists, or not, we can go ahead with the rest of the validation of input. Validation ranges from checking string length, checking whether password & password confirmation match and whether the address from **Figure 24** is full. As you may notice, I have used flash to give the user feedback with custom messages. Once all this validation is complete, we can move on with the rest of the registration process.

```
# Create a database instance
cursor = CONNECT.cursor()
# Check if the email already exists in our database
query = cursor.execute(
    '''SELECT email FROM seniors WHERE email = '%s' ''' % email)
# If a result returns, set our pre-existing variable to True
if query != 0:
    email_exists = True
cursor.close()

# Create some conditions and checks
if len(email) < 4:
    flash('Email must be greater 3 characters', category='error')
elif email_exists:
    flash('Email already exists!', category='error')
elif len(first_name) < 4:
    flash('First Name must be greater than 3 characters', category='error')
elif len(family_name) < 4:
    flash('Family Name must be greater than 3 characters', category='error')
elif password != password_confirm:
    flash('Passwords do not match', category='error')
elif len(password) < 8:
    flash('Password must be greater than 7 characters', category='error')
elif full_address_complete is False:
    flash('Address must not have empty fields', category='error')
else:
```

Figure 26: Registration input validation and verification

```
cursor = connect.cursor()
# Populate the usergroups table if the information isn't there
cursor.execute('''
    INSERT IGNORE INTO usergroups
        (group_id, description)
    VALUES
        (1, 'Administrator'),
        (2, 'General User'),
        (3, 'Social Member'),
        (4, 'Senior Member'),
        (5, 'Senior Member in Full Time Education or Unemployed')
''')
connect.commit()
cursor.close()
```

Figure 27: Populating the 'usergroups' table

At this point, I thought it would be a good idea to populate the 'usergroups' database table because upon registration, I want to assign the user a group when I go to insert their details into the 'seniors' table. Heading back to 'main.py', as previously shown in **Appendix B (p. 54)**, I can append the table population process to the bottom of the rest of the database table checks. See **Figure 27** on the left

Continuing the registration process (**Figure 28**), we want to ensure the user has a secure account on our end. Using a dependency called 'werkzeug.security', we can generate a password hash using SHA256 which will protect the password because it cannot be decrypted. Now we are ready to insert all this data into our seniors table. It's all verified and ready to go. Commit the changes and close the connection. We also provide feedback to the user in the form of a flashed message informing them that their account creation was a success. The working registration page can be found in **Appendix E (p.57)**.

```
    password_hash = generate_password_hash(password, method='sha256')
    # Create a database instance
    cursor = CONNECT.cursor()
    # Insert all our user data into the database
    cursor.execute(
        '''INSERT INTO seniors (first_name, family_name, email, password, address, phone_number)
        VALUES ('%s', '%s', '%s', '%s', '%s', '%s')'''
        % (first_name, family_name, email, password_hash, full_address, phone_number))
    # Commit our new change
    CONNECT.commit()
    cursor.close()
    flash('Account successfully created!', category='success')
return render_template("register.html")
```

Figure 28: Completing the registration process

5.5 Login

The login page, as laid out in the Login page storyboard (**Figure 3**), will consist of a simple POST request form asking for an email and password combination as shown in **Figure 29**.

```
{% extends "base.html" %} {% block title %}WGCCC - Login{%endblock%}
{%block content%}
<form method="POST">
    <h3 align="center"> Login </h3>
    <div class="form-group">
        <label for="email">Email Address</label>
        <input
            type="email"
            class="form-control"
            id="email"
            name="email"
            placeholder="Enter your email..."
        />
        <label for="password">Password</label>
        <input
            type="password"
            class="form-control"
            id="password"
            name="password"
            placeholder="Enter a secure password..."
        />
</div>
<br />
<button type="submit" class="btn btn-primary auth-submit">Login</button>
{%endblock%}
```

Figure 29: Creating a POST request form for the login page

The login page back-end (**Figure 30**), alongside the registration page logic, sits in 'auth.py'. To start off, we want to retrieve the POST request data and assign it to some variables. We want to check if the email provided exists in the first place by querying the database and comparing the stored value and the value given by the user. We will fetch the first row only and close the connection. If a user doesn't exist, we will flash an error and have them start again. If a user does exist, we will use an in-built function from 'werkzeug.security' to check if the password hashes match and flash a success or an error (**Appendix F (p. 58)**).

```
@auth.route('/login', methods=['GET', 'POST'])
def login():
    # Get the POST request content
    if request.method == 'POST':
        email = request.form.get('email')
        password = request.form.get('password')

        cursor = CONNECT.cursor()

        # Retreive the user if the email exists in the database
        cursor.execute(
            '''SELECT senior_id, password FROM seniors WHERE email='%s' '''
            % email
        )
        user = cursor.fetchone()
        cursor.close()

        # We have a valid user
        if user:
            # Check the password hash input matches the password hash stored
            if check_password_hash(user[1], password):
                # User has successfully logged in
                flash('Logged in Successfully!', category='success')
                return redirect(url_for('views.home'))
            else:
                # User entered an incorrect password and forced to try again
                flash('Incorrect password. Please try again!', category='error')
        # User doesn't exist in the database
        else:
            flash('Incorrect email. Please try again!', category='error')
    return render_template("login.html", DATETIME=str(datetime.now().year))
```

Figure 30: Login page back-end

5.6 Sessions and Persistence

Upon further research of Flask (n.d.) Documentation, I discovered that I could save data to something called a Session. Importing the Session dependency through Flask means that I can use data from one page to create global variables – accessible through all parts of the application. This changes the way my application works altogether and opens lots of opportunities. Sessions can be viewed by the client but cannot be edited unlike a traditional cookie. However, this is reliant on the secret key I assigned in my environment file. If the user knows this key, they can make modifications. With this new knowledge I can begin to create new session variables upon login to keep track of the user, their details, and their status. See **Figure 31**. I have set session persistence to have a lifespan of 7 days in 'main.py'. This means that any session variables I set won't be wiped until this time limit. In 'auth.py', I

can save details of choice to the session. I have also created a flag to determine whether a user is logged in. When setting 'session.permanent', it determines whether it follows the session lifetime in 'main.py' so I must declare it.

main.py

```
# Set Session persistence to a specified amount of days
app.permanent_session_lifetime = timedelta(days=7)
```

base.html | login.html | views.py | auth.py | register.html

```
if user:
    # Store their relevant information in variables from the db query
    user_id = user[0]
    user_password_hash = user[1]
    user_name = user[2]
    phone_number = user[3]

    # Check the password hash input matches the password hash stored
    if check_password_hash(user_password_hash, password):
        # Store the session variables and keep persistence
        session.permanent = True
        session['loggedin'] = True
        session['email'] = email
        session['name'] = user_name
        session['id'] = user_id
        if phone_number is not None:
            session['phone'] = phone_number
        # User has successfully logged in
        flash('Logged in Successfully!', category='success')
```

Figure 31: Setting Session variables on login and persistence

5.7 Logout

The user should have the choice to logout themselves rather than wait for the session to expire. I have created a 'logout' route that has no visual page but will execute code upon going to the path or when redirected to. In **Figure 32**, if a user tries to go to the link without being logged in, they will receive feedback and be redirected to the login page. If not, all the session variables will be cleared, and they will be logged out.

```
@auth.route('/logout')
def logout():
    # Clear all session variables and redirect home if user is logged in
    if 'loggedin' not in session:
        flash("You are not logged in!", category='error')
        return redirect(url_for('auth.login'))

    session.clear()

    return redirect(url_for('views.home'))
```

Figure 32: Logout and clear all session variables

5.8 Subscriptions

The Subscriptions page, as outlined in the storyboard (**Figure 4**), should consist of separate products having their own product cards. As most of the other pages, it will be a POST form that will send data to the server with a corresponding type of subscription which we can handle on the back-end. Please see **Figure 33**. The information we want to display is hardcoded at the moment, but is fine for our immediate requirements considering there is an unlikely chance of change. It's not a high priority.

```
{% extends "base.html" %} {% block title %}WGCCC - Subscriptions{%endblock%}
{%block content%}
    <div class='seniors'>
        <h1>Senior Subscriptions</h1>
        <div class="product-container">
            <ul class='product-card'>
                <li class='name'>Senior Member</li>
                <li class='short_desc'>*social membership included</li>
                <li class='price'>£180.00</li>
                <li>
                    <form name='senior' method="POST">
                        <input type="hidden" id="senior" name="senior" value="senior">
                        <button id='checkout' class="btn btn-primary" type="submit">Checkout</button>
                    </form>
                </li>
            </ul>
            <ul class='product-card'>
                <li class='name'>Senior Member</li>
                <li class='short_desc'>(Unemployed / Full-Time Education)</li>
                <li class='price'>£90.00</li>
                <li>
                    <form name='senior_edu' method="POST">
                        <input type="hidden" id="senior_edu" name="senior_edu" value="senior_edu">
                        <button id='checkout' class="btn btn-primary" type="submit">Checkout</button>
                    </form>
                </li>
            </ul>
```

Figure 33: Creating 'subscriptions.html' with separate product cards in a POST form

```
// Modal implementation used on the Subscription page
var modals = document.getElementsByClassName('modal');
var btns = document.getElementsByClassName("openmodal");
var spans = document.getElementsByClassName("close");

for(let i=0;i<btns.length;i++){
    btns[i].onclick = function() {
        modals[i].style.display = "block";
    }
}

for(let i=0;i<spans.length;i++){
    spans[i].onclick = function() {
        modals[i].style.display = "none";
    }
 }

 window.onclick = function(event) {
     for(let i=0;i<modals.length;i++){
         if (event.target == modals[i]) {
             modals[i].style.display = "none";
         }
     }
 }
```

Figure 34: Modal JavaScript

The Subscriptions page consists of both Senior and Junior subscriptions. However, we need to collect more information about the Junior to proceed with the purchase process should a prospective customer wish to purchase one. For this, I decided to implement a pop-up modal because it's less intrusive than switching page content. I used inspiration from W3Schools (n.d.) for this implementation. Please see **Appendix G (p. 59)** to view how I implemented a modal for both the Junior Subscription and Junior (Developmental) Subscription in HTML. The modal also requires some underlying JavaScript found in **Figure 34.** Upon the user clicking any button named 'openmodal' the modal display attribute will show, otherwise if the close button is clicked, or the window surrounding the modal, it will close, and the display attribute is set to 'none'.

5.9 Stripe API Implementation

The implementation of the Stripe API will mostly reside in 'views.py', in the Subscriptions route. We want to handle all our data collection and process of payments in the subscriptions section of the application. To start off we should access Stripe Dashboard (https://dashboard.stripe.com/) and do some configuration once an account is created.

5.10 Stripe Dashboard

The Stripe dashboard offers a very useful test mode which means that you don't need to spend real money to test the product or application. After setting up an account, I need to setup some new products. It will also hold information about active subscriptions, customer history, and much more.

Although I stated that I would hold this information in my own database, I want to hold it concurrently both third-party and in self-storage for now to ensure future considerations. Time constraints restrict me from creating my own payment pages using the Stripe API, so using the dashboard is a necessity. To interact with the Stripe API, a private API key is provided to the Stripe account. I have saved this API key in the .env file for use when needed.

Setting up the products was easy and only needed a type (subscription), name, price, and period of billing. For my requirements, a yearly subscription was set up with various prices referenced from the original cricket club website. See **Figure 35** for a list of the products.

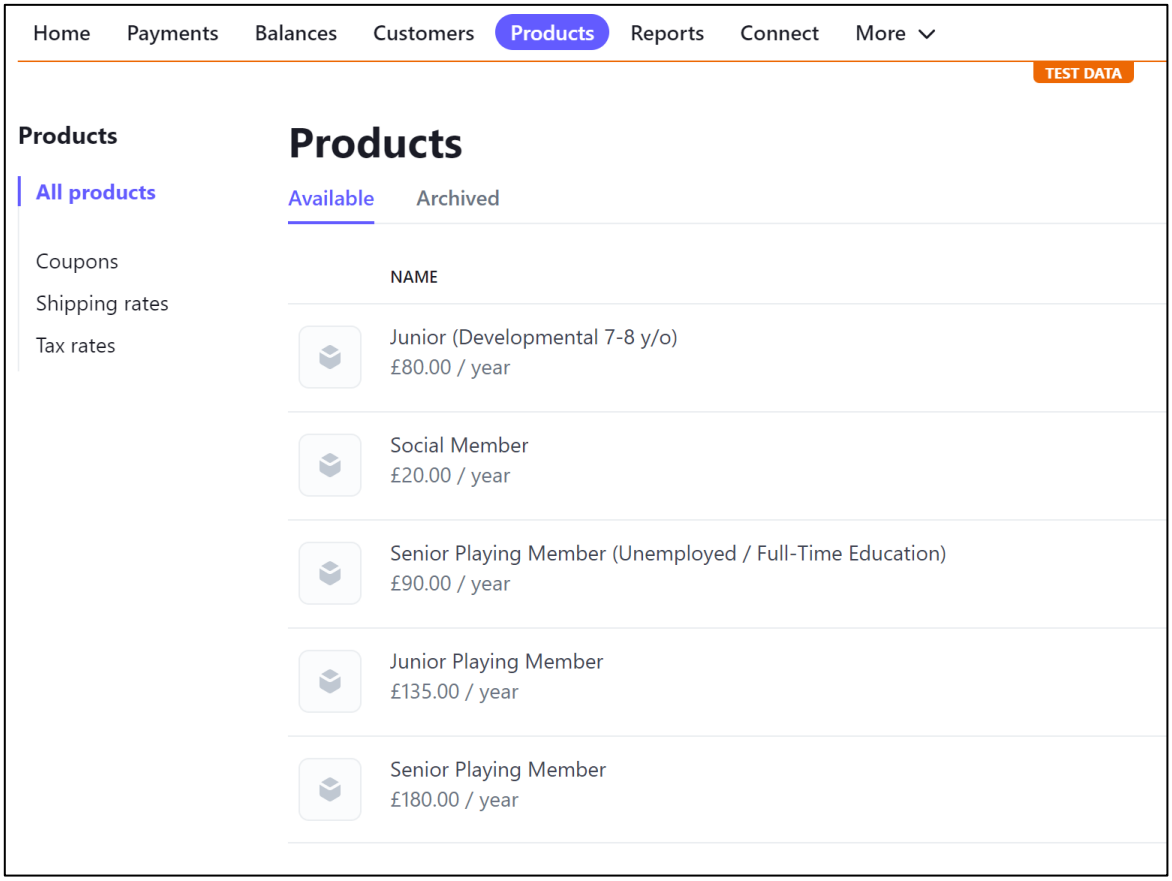


Figure 35: Product list and their prices on Stripe Dashboard

When setting a price on a product, it's given a unique Price ID. This is an ID we can use to determine what product is what when implementing the API into the application.

5.11 Stripe Payment/Checkout Session

According to Stripe (n.d.) Documentation, a Stripe Payment/Checkout Session is a "secure, Stripe-hosted payment page that lets you collect payments quickly". This is vital for my project especially considering time-constraints and as previously mentioned, will save me from having to create my own payment page.

At first, I thought I would need to make a Stripe Checkout Session for each condition (conditions being whichever subscription the user picks). Upon further thought, I thought it better to store some information in a Python Dictionary including a group_id from the database, the price_ids for each product and a flag to help me determine the kind of queries I will need to run, taking into consideration whether a subscription is for a junior or a senior. From the POST request, I can grab the data to determine what product has been chosen. **Figure 36** demonstrates the dictionary and **Figure 37** demonstrates how I created a loop to match the POST data with the values in the dictionary.

```
# Create a dictionary with subscription information and flags
# Key: name value: [group id, price id, flag type]
price_dict = {
    'social': ['3', 'price_1KTT07HuaTKPzffSkYKw4EPw', 'senior_flag'],
    'senior': ['4', 'price_1KTNDVHuaTKPzffS1ubgGAr7', 'senior_flag'],
    'senior_edu': ['5', 'price_1KTS4HHuaTKPzffSsYTpJcNQ', 'senior_flag'],
    'junior': [None, 'price_1KTOPyHuaTKPzffS5yvO1LSb', 'junior_flag'],
    'junior_dev': [None, 'price_1KTnk8HuaTKPzffSo8JBgFX2', 'junior_flag']
}
```

Figure 36: Python Dictionary storing values dependent on the product html element key

```
price_id = None
for k, v in price_dict.items():
    if request.form.get(k):
        price_id = v[1]
if price_id is None:
    flash(
        # Products are misconfigured and need manual changes
        "There was an error - please contact the system administrator",
        category='error')
    return redirect(url_for('views.home'))
```

Figure 37: Looping through the dictionary to assign a variable with the matching price id

As seen in **Figure 37**, I primarily initialise a variable named 'price_id'. I have looped through the dictionary getting both the keys and the values. Within the loop, I have a condition that checks whether the POST data matches one of the keys and assigns the second index value to 'price_id'. If nothing is found, it just drops out and calls another condition to check if 'price_id' is still None. If so, then the products are misconfigured and the user cannot proceed with payment of a subscription.

```
# Create a new stripe session to proceed with payment
stripe_session = stripe.checkout.Session.create(
    customer_email=session['email'],
    line_items=[{
        'price': price_id,
        'quantity': 1
    }],
    mode='subscription',
    success_url=url_for('views.home'),
    cancel_url=url_for('views.subscriptions')
)
```

Figure 38: Creating a Stripe Checkout Session

Now that we have the product we want, we can create the Stripe Checkout Session Object. As shown in **Figure 38**, I have assigned the price id as our line item and assigned the session to the user's email which we can grab from our app session. It also contains a 'success_url' and a 'cancel_url'. These are pages that the session will redirect to on success of payment and cancellation of payment respectively.

The Senior memberships don't require any kind of validation on the subscriptions page – it's a different matter for Junior memberships.

At this point I have decided to add a 'date of birth' field to my 'juniors' database table in case I wish to make additions or changes to how the modal works in the future. Time-constraints force me not to take make the change now, but this will be discussed in the future considerations section of this report.

As mentioned previously and as shown in **Appendix G (p.59)**, a modal pops up when trying to select a Junior Membership with details the user must fill out before continuing. We need to validate this information in **Figure 39.** As demonstrated, we loop through the dictionary again and perform a check whether the key-value pair holds a 'junior_flag' AND the 'price_id' is a junior price id. Both conditions must be met to continue. Let's assume both conditions are met, we want to assign some variables with the POST request data and perform some checks. If they do not meet the check, the user is informed, and we drop out of the loop. Otherwise, we can save our data into some session variables.

```
# Loop through our dictionary and do some conditions and checks
for k, v in price_dict.items():
    # If a key has a Junior Flag and the price_id matches our...
    # ...session, get POST content and assign them to some variables
    if v[2] == 'junior_flag' and v[1] == price_id:
        first_name = request.form.get('junior_first_name')
        family_name = request.form.get('junior_family_name')
        birthdate = request.form.get('junior_dob')
        if len(first_name) < 4:
            flash('First Name must be greater than 3 characters',
                  category='error')
            break
        elif len(family_name) < 4:
            flash('Family Name must be greater than 3 characters',
                  category='error')
            break
        elif birthdate == '':
            flash('Please insert a Date of Birth', category='error')
            break
        elif birthdate > str(datetime.now()):
            flash('Date cannot be in the future', category='error')
            break
        else:
            # All conditions have passed, create new session...
            # .. variables to be used on the Success page
            session['junior_first_name'] = request.form.get(
                'junior_first_name')
            session['junior_family_name'] = request.form.get(
                'junior_family_name')
            session['junior_dob'] = request.form.get('junior_dob')
# Redirect to the third-party Stripe Session page
return redirect(stripe_session.url, code=303)
```

Figure 39: Validation of input in Junior Membership modal

Finally, we can be redirected to Stripe's payment page with the Stripe Session URL. Thankfully for the test suite, it provides faux card details that you can test with. See **Figure 40**. As you can see, the email has been assigned automatically and the product details are on the left.

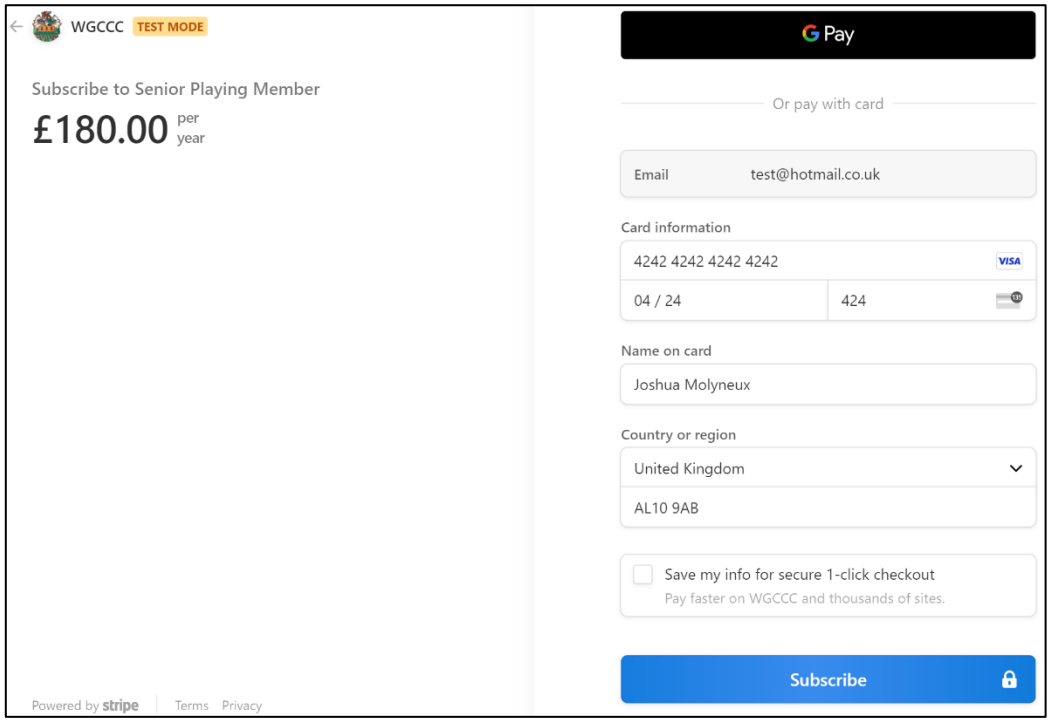


Figure 40: The Stripe Payment Session page

5.12 Success Page/Route

The Stripe Checkout Session, as mentioned previously in **Figure 38**, directs the user to a given URL when the payment is successful. By default, I had left it to only redirect back to the home page. However, I want to add some information to be stored in the database if a successful payment is made by the user.

To do this, I first made small changes to the way the Stripe Checkout Session works. In **Figure 41**, I first stored the id of the Stripe Session object and then changed the success URL to pass the Stripe Session ID and Price ID of the product that was being purchased. I also thought it would be a good idea to put the user's id into the metadata in-case of any issues. I did run into an issue during development for URL usage when testing both live and locally. So, I had to create some checks and variables to cater for it.

```
# DEVELOPMENT PURPOSES
url = urlparse(request.base_url)
hostname = url.hostname
port = ''
if hostname == 'localhost':
    port = ':5000'
else:
    port = ':80'
# Create a new stripe session to proceed with payment
stripe_session = stripe.checkout.Session.create(
    customer_email=session['email'],
    line_items=[{
        'price': price_id,
        'quantity': 1
    }],
    metadata={'user_id': session['id']},
    mode='subscription',
    success_url='http://%s%s/success?session_id={CHECKOUT_SESSION_ID}&price_id=%s'
    % (hostname, port, price_id),
    cancel_url='http://%s%s/subscriptions'
    % (hostname, port)
)
# Create a new session variable with the id of the Stripe Sesssion
session['stripe_session'] = stripe_session.id
```

Figure 41: Modifying the Stripe Checkout Session object

Now we will create a new route named '/success' and conduct some preliminary checks to make sure users aren't accessing the page without going through the correct process of payment. See **Figure 42.** Firstly, we check if the 'stripe_session' session variable even exists on the server, then check if the URL path contains the session_id, and then finally check whether our variable stored on the server matches the session_id argument in the URL. If one of these conditions doesn't meet, the user is flashed with an error, denied access and redirected home.

```python
@views.route('/success')
def success():
    if 'loggedin' not in session:
        flash("You are not logged in!", category='error')
        return redirect(url_for('auth.login'))
    # Conditions too long for one line, split them up a bit for readability
    check_session_exists = 'stripe_session' not in session
    check_id_in_args = 'session_id' not in request.args
    check_session_query = session['stripe_session'] != request.args.get(
        'session_id')
    # We don't want users to access the success page without actually buying...
    # ...anything
    if check_session_exists or check_id_in_args or check_session_query:
        flash("You cannot access this page from here", category='error')
        return redirect(url_for('views.home'))
```

Figure 42: Creating a new 'success' route with some checks to stop direct access

Now that we have conducted some checks for valid access, I can start adding data to the database and execute queries dependent on what the subscription type was. For reference, please see **Figure 43**. We, again, loop through our price dictionary and check whether the payment was for a senior subscription and which product it was. If it's the case we can update the user with their new group_id.

```python
id = session['id']
# Create a database instance
cursor = CONNECT.cursor()
# Loop through our dictionary again
for k, v in price_dict.items():
    # Check if the subscription is a senior one and grab the arg from...
    # ..the success url
    if v[2] == 'senior_flag' and v[1] == request.args.get('price_id'):
        # Update the user's group with the corresponding ID from the dict
        cursor.execute('''
            UPDATE seniors
            SET group_id = '%s'
            WHERE senior_id = '%s'
        ''' % (v[0], id))
        # Commit the change
        CONNECT.commit()
        cursor.close()
```

Figure 43: Updating the user in the 'seniors' table

Handling the Junior subscription payment is slightly more complicated. We need to first check whether it was even a junior subscription in the first place, but then we must determine what kind of junior subscription it was. Please see **Figure 44** for reference. After checking whether it's a junior subscription, we can set some variables from the junior session variables. Dependent on the key from the dictionary, I carry out queries, the only difference being that the 'is_developmental' field is set to 1 if the key is 'junior_dev'. We have done all we need to do with the junior session variables so we can destroy them.

```
# If the subscription is a junior one and the price id matches...
elif v[2] == 'junior_flag' and v[1] == request.args.get('price_id'):
    # ...set some variables from our session variables
    junior_first_name = session['junior_first_name']
    junior_family_name = session['junior_family_name']
    junior_dob = session['junior_dob']

    # If the key is the junior subscription...
    if k == 'junior':
        # ...Insert our session data into the database
        cursor.execute('''
            INSERT INTO juniors
                (first_name, family_name, dob, senior_id)
            VALUES
                ('%s', '%s', '%s', '%s')
        ''' % (junior_first_name, junior_family_name, junior_dob, id))
        # Commit changes
        CONNECT.commit()
        cursor.close()
    # If the key is the developmental junior subscription...
    elif k == 'junior_dev':
        # Insert data into the datbase and set 'is_developmental' to 1
        cursor.execute('''
            INSERT INTO juniors
                (first_name, family_name, dob, senior_id, is_developmental)
            VALUES
                ('%s', '%s', '%s', '%s', 1)
        ''' % (junior_first_name, junior_family_name, junior_dob, id))
        # Save changes
        CONNECT.commit()
        cursor.close()
    # Garbage collection
    session.pop('junior_first_name', None)
    session.pop('junior_family_name', None)
    session.pop('junior_dob', None)
```

Figure 44: Handling the successful Junior subscription payment

6.0 Code Revisions

6.1 International Phone Number support

In preparation for the implementation of SMS notifications, I thought it would be a good idea to increase the input validation of phone numbers by reducing human error when it comes to international phone numbers or country codes. Using inspiration from a user named 'miguelgrinberg' (2019) on GitHub.com, I was able to implement an input box in JavaScript on the registration page. Please see **Figure 45** and **Figure** 46 for reference. The script utilises the 'intl-tel-input' library and alters the input box to display flags and country codes next to the input box, along with a drop-down menu of different countries.

```
// International country code implementation
// Inspired by https://github.com/miguelgrinberg/flask-phone-input/blob/master/templates/index.html
 var phone_field = document.getElementById('phone');
 phone_field.style.position = 'absolute';
 phone_field.style.top = '-9999px';``
 phone_field.parentElement.insertAdjacentHTML('beforeend', '<div><input type="tel" class="form-control" id="_phone"></div>');
 var intl_phone_field = document.getElementById('_phone');
 var intl_phone_iti = window.intlTelInput(intl_phone_field, {
     initialCountry: 'gb' ,
     separateDialCode: true,
     utilsScript: "https://cdnjs.cloudflare.com/ajax/libs/intl-tel-input/16.0.4/js/utils.js",
 });
 intl_phone_iti.setNumber(phone_field.value);
 intl_phone_field.addEventListener('blur', function() {
     phone_field.value = intl_phone_iti.getNumber();
 });
```

Figure 45: International phone input

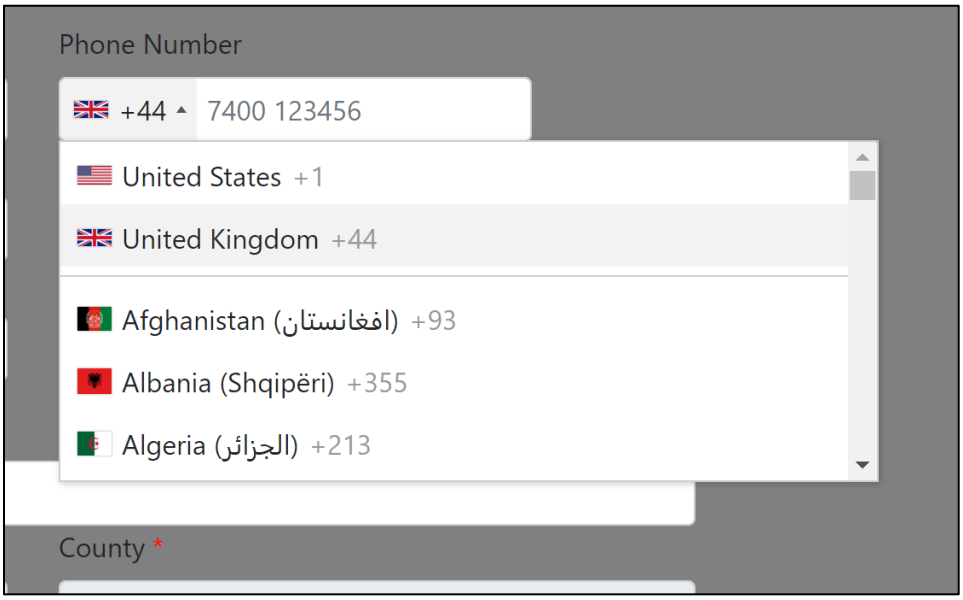


Figure 46: Updated phone input on registration page

6.2 SMS Push Notification

As laid out in my Project Plan, I mentioned that I would follow a MoSCoW approach to my project. The SMS Push Notification feature was pushed quickly as a 'Should have' and doesn't offer a lot in terms of functionality right now however, it performs a purpose of assuring the user of a successful payment. TextMagic offers an API key which I have placed in my .env file for security purposes along with my TextMagic username. From the TextMagic (n.d.) Rest API Documentation, I was able to gain inspiration on how to use the Python wrapper and implemented it into my own work. It made the most sense to implement this on the success page.

In **Figure 47**, you can see that I have retrieved both the product and its price from the stripe session using the ID I saved to a session variable. The price however, comes unformatted so I had to do that myself using the 'format()' function (e.g. a price of £180, would show as 18000). We no longer need the session variable so we can garbage collect it. I have assigned some variables with two session variables of the user's name and ID. Next, I wanted to be sure that a phone number exists as a session variable, which was set when the user logs in. The international phone input has a '+' in front of the number when collecting input, so we need to strip that before we can proceed. Using the TextMagic API, we can then attempt to send a message to the phone number and just pass through if it was unsuccessful.

```
# Retrieve both the product, and its price from the Stripe Session
item = stripe.checkout.Session.list_line_items(session['stripe_session'])
product = item.data[0].price.product
product = stripe.Product.retrieve(product).name
price = item.data[0].price.id
price = stripe.Price.retrieve(price).unit_amount
# Price comes unformatted, let's fix that
price = format(price / 100, '.02f')

# We no longer need the Stripe Session, garbage collect
session.pop('stripe_session', None)

# Assign some variables from our session that we plan to use
id = session['id']
name = session['name']

# SMS Notification of successful payment
# Check if the user even has a phone number connected to the account
if 'phone' in session:
    # Try to send a message (it ignores the condition for some reason)
    try:
        phone = session['phone'].replace('+', '')
        print(phone)
        client = TextmagicRestClient(TEXTMAGIC_USERNAME, TEXTMAGIC_API_KEY)
        client.messages.create(
            phones=phone,
            text=f"Hi, {name}. Your purchase of '{product}' for £{price} was successful!"
        )
    except:
        # Send was unsuccessful, just carry on
        pass
```

Figure 47: Sending a successful payment SMS to the user

7.0 System/Requirements Testing

Testing the system and its requirements is an important step in the development process to ensure system success and allows evaluation of areas for improvement. The system specification the system tests were produced on are:

Table 5: A table of specifications used to test the system

CPU	Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
GPU	Intel UHD Graphics 620
RAM	8.00GB
Operating System	Windows 10 Pro
SSD Storage	256GB
Network Speed	Down: 115Mbps / Up: 22Mbps

An internet connection is required.

7.1 System Profiling

Using a Profiler provided by the 'werkzeug' library, I can gauge resource-expensive pages/functions. After conducting some profiling tests on each page, it's clear which loading times are far worse than others. However, some are miniscule that have no effect on the user experience.

Table 6: A table to show pages and their load times in milliseconds

Request and Page	Response Time
GET:root (first load)	517ms
GET:root (subsequent load)	3ms
GET:register	5ms
POST:register	661ms
GET:login	6ms
POST:login	38ms
GET:logout	2ms
GET:subscriptions	3ms
POST:subscriptions	639ms

As shown from the table, it's evident and as expected, that pages performing a POST request will take far longer to get a response than those which are performing GET requests. This is because POST requests are sending data to the server. The first load of root, regardless of being a GET request, is time-consuming because it's running initialisation steps and running the 'before_first_request_func()' in Main.py.

7.2 Requirements Testing

Testing the functional and non-functional requirements set out in the design phase of this project and laid out in **Table 3 & 4 (p. 15)** is conducted to assess whether requirements have been met and overall demonstrate the success of the design.

Table 7: A table to show requirements testing and mitigation/changes

Expected requirement	Result	Mitigation or Changes (if applicable)
A website user should be able to login and access available subscriptions	Anyone can login and view available subscriptions	
A website user should be able to view subscriptions so that they can purchase one	Only someone with a registered account can purchase one	
A Club Official should be able to access existing customers to monitor activity	Through Stripe Dashboard, Club Officials will be able to access existing customers	Changes could be made to develop an administrative panel without the use of a third-party panel/dashboard
A website user should be able to access their profile to view and edit their information	Users can't access their profile or information	Develop a profile page, giving the users an option to make changes to their information
A customer should receive a push notification to keep track of invoices	A short SMS is sent upon a customer purchasing a subscription	Emails would also be sent and be more detailed than it is currently
A Club Official should be able to manage available subscriptions so that products can stay up to date should they need changing	Through Stripe Dashboard products can be added and removed	As mentioned, changes could be made to create an administrative panel.
An existing customer should be able to access their profile to view active and past subscriptions	Profile page has not been developed; however, SMS will send a text with the subscription and price.	Development of a profile page
A website user should be able to input data about a junior so they can purchase a junior subscription	The user can input a name and date of birth about a junior which is stored in the database if a successful payment is made	
A website user should navigate a clear sitemap for ease-of-use	The website has a clear sitemap due to its current simplicity	If further development is made, the sitemap should remain simple to use
A website user should be viewing webpages with a suitable colour-scheme	The website bares the same colour on all pages except for when they are redirected to the Stripe Checkout Session which is completely different	As mentioned, changes made would be creating my own checkout page and handling payment whilst still utilising Stripe API but at a lower level
A website user should experience high loading speeds on all pages for a smooth experience	As per the profiler data, loading speeds vary depending on the HTTP requests being sent but are not very noticeable after first load	Some code clean-up and simplifications in logic could be made

8.0 Conclusions & Considerations

The main aim of this project was to experiment with a solution to the problem of a lack of centralised system to manage payments for Welwyn Garden City Cricket Club memberships. The undertaking of this project proved that there is a way to get around this issue albeit, in a way that costs a few more financial overheads in the form of Stripe taking a cut of the payment, and TextMagic charging for SMS when sending messages to users. The 'notifications' objective of my project was not fully experimented with and more could have been done to assess the effectiveness of such a feature in conjunction with the payments system, although stated throughout the report that it would be a 'should have'. I am still satisfied that I managed to gain insight on the subscriptions implementation with a centralised area being the Stripe Dashboard using Python as a back-end to establish a web-interface, which I didn't know was possible before conducting research for this project.

I faced some difficulties throughout the project such as struggling to figure out how to process the different subscriptions when the user goes to pay on the 'subscriptions' and 'success' pages in auth.py. In hindsight, I feel like my code is quite inefficient and unmaintainable in this regard. The global dictionary, 'price_dict' (**Figure 36 (p. 39)**), with hardcoded values could be optimised by interfacing the Stripe API itself and grabbing the price id values from the products I have created. Another difficulty I came across is my creativity on the front-end. When producing my artefact, I focussed mostly on building the back-end to a good standard rather than making the design look aesthetically sound. I do believe this to be just a personal issue with creative weakness rather than an issue with the project itself. Another difficulty faced, was customer creation on the Stripe Dashboard. Every time a subscription is bought, a new customer is made, regardless whether the customer has an existing subscription. It's only after the fact, that I now realise I would need to change the way I create the Checkout Session. After doing some further research on the Stripe (n.d.) Documentation regarding 'The Session Object', I found how to mitigate this. The way I thought I was supposed to do it may have been incorrect but, I was already passed the point of amending my code to test my theory. Another difficulty I faced is trying to get my database to automatically create itself (**Figure 14 (p. 27)**). Sometimes it worked and sometimes it didn't. I'm unaware of how to fix this issue, albeit it was more of a quality-of-life implementation. Some of these difficulties are to be discussed as a future consideration.

In general, I felt that the project went well. I've learnt a few things about the process of software development that I felt I needed more work on. It's been a huge eye-opener for me in terms of understanding the potential I can achieve and what I can learn when I set my mind to a project like this. I've come to appreciate how important the design phase in a project can be, especially when it comes to deciding how to approach an issue and laying out a map of requirements to fulfil. Some of the design work I produced did not reach the production phase unfortunately due to prioritisation on subscriptions, such as the home page or profile page. However, I will go into further discussion about this in my future considerations.

8.1 Future Considerations

My future considerations are aspects, features, and details of my artefact that I feel are necessary in future development to create an effective piece of software.

First and foremost, I think it's important to address features that weren't implemented even though they were set out in the design of my artefact. The home page right now is blank. Nothing was implemented that I set out to do in this regard, except of course for the global banners, logo, and navigation. The home page, in future development, will attempt to mimic the design I created in **Figure 1 (p. 16)**. Another storyboard I failed to complete was the Profile Page. The profile page is a vital part of the system that allows a user to edit their personal information such as name, password, and email. In future consideration, this would be a priority and would also attempt to mirror its storyboard design in **Figure 5 (p. 20)**.

The next consideration would be re-coding the logic that determines what subscription has been selected in auth.py utilising the 'price_dict' global dictionary. It feels incredibly inefficient and could be improved. As stated before, I would likely initialise a dictionary through the database and Stripe API instead of hardcoding information. The issue with this is that it would require changes to the codebase itself in the future should a product be added or removed.

Another consideration to be made is the way a Stripe Checkout Session is created. A customer is created each time a subscription is bought, even if bought by the same customer multiple times. The object creation would need to be changed to specify the customer's ID which would reference the Customer Object rather than the user's email.

Email notifications and invoices are something that I wanted to implement alongside SMS notifications, however due to time constraints it no longer became a possibility. Creating an SMTP server for automated mail would assist in this effort. Sending invoices rather than a simple SMS by itself would be an ideal outcome to help users keep track of what they have purchased.

In relation to Stripe Customer Object creation, Customers, right now, can buy multiple senior memberships at a time. I don't believe this should be possible for the case of mitigating human error. They should, however, continue to be able to buy multiple junior subscriptions. To further automation efforts, subscriptions should change dependant on changing factors such as juniors no longer being a developmental junior. This could work with SMS or Email informing the user that their subscription is changing with notice.

A penultimate consideration should be that the subscriptions that are bought are not saved to the in-house database. It's something I wanted to do for extensibility reasons and would save time in the future instead of a data export from Stripe Dashboard should considerations to redesign how payments are processed in the future are made if the Stripe Checkout Session is to be replaced.

The final consideration to be made is to push my artefact to a production phase on a Port 443 web server to utilise HTTPS. I inferred in my literature review that it would be an important aspect of the artefact, however I had no need to do this as my artefact is not complete yet.

9.0 References

A. Moka, J (2020) *Data as Commodity: For Data Science Professional* [online]
Available at: https://www.datasciencecentral.com/profiles/blogs/data-as-commodity-for-data-science-professional
[Accessed 3rd November 2021]

Big Commerce (n.d.) *What You Need to Know About Securing Your Ecommerce Site Against Cyber Threats* [online]
Available at: https://www.bigcommerce.co.uk/articles/ecommerce/ecommerce-website-security
[Accessed 7th November 2021]

Baker, S (n.d.) *Payment Gateway vs Processor: What is the Difference?* [online]
Available at: https://financesonline.com/payment-gateway-processor-difference
[Accessed 3rd November 2021]

Darragh, R (2021) *The 7 Best Payment Gateways and Online Payment Systems 2021* [online]
Available at: https://startups.co.uk/payment-processing/best-payment-gateways
[Accessed 6th November 2021]

Digital.com (2021) *Best Web Hosting Security Practices* [online]
Available at: https://digital.com/best-web-hosting/security-practices
[Accessed 8th November 2021]

Flask (n.d.) *About the First Parameter* [online]
Available at: https://flask.palletsprojects.com/en/2.1.x/api/#flask.Flask
[Accessed 11th April 2022]

Flask (n.d) *Message Flashing* [online]
Available at: https://flask.palletsprojects.com/en/2.1.x/patterns/flashing
[Accessed 11th April 2022]

Flask (n.d.) *Sessions* [online]
Available at: https://flask.palletsprojects.com/en/2.1.x/api/#sessions
[Accessed 11th April 2022]

GeeksforGeeks (2021) *Frontend vs Backend* [online]
Available at: https://www.geeksforgeeks.org/frontend-vs-backend
[Accessed 3rd November 2021]

Goree, S (2020) *Yes, websites really are starting to look more similar* [online]
Available at: https://theconversation.com/yes-websites-really-are-starting-to-look-more-similar-136484
[Accessed 12th April 2022]

HealthIT.gov (2019) *What does "https" in a web address mean?* [online]
Available at: https://www.healthit.gov/faq/what-does-https-web-address-mean
[Accessed 8th November 2021]

Hendricks, D (n.d.) *10 Essential Steps to Improve Your Website Security* [online]
Available at: https://www.computer.org/publications/tech-news/trends/10-essential-steps-to-improve-your-website-security
[Accessed 7th November 2021]

IBM (2020) *Application Programming Interface (API)* [online]
Available at: https://www.ibm.com/cloud/learn/api
[Accessed 4th November 2021]

IBM (2021) *How much does a data breach cost*? [online]
Available at: https://www.ibm.com/security/data-breach
[Accessed 8th November 2021]

ICO (n.d.) *Children and the GDPR* [online]
Available at: https://ico.org.uk/media/for-organisations/guide-to-the-general-data-protection-regulation-gdpr/children-and-the-gdpr-1-0.pdf
[Accessed 12th April 2022]

Kansal, A (n.d.) *The Complete Guide to Website Push Notifications for E-commerce* [online]
Available at: https://neilpatel.com/blog/website-push-notifications-for-ecommerce
[Accessed 7th November 2021]

Miguelgrinberg (2019) *flask-phone-input* [online]
Available at: https://github.com/miguelgrinberg/flask-phone-input/blob/master/templates/index.html
[Accessed 12th April 2022]

Oriyano, S-P (2016) *Certified Ethical Hacker Version 9 Study Guide* [eBook]
Available at: https://books.google.co.uk/books?id=AEwIDAAAQBAJ
[Accessed 5th April 2022]

Stack Overflow (2015) *Combine street number and route in Address autocomplete api* [online]
Available at: https://stackoverflow.com/questions/31364880/combine-street-number-and-route-in-address-autocomplete-api
[Accessed 30th March 2022]

Stripe (n.d.) *Stripe Checkout* [online]
Available at: https://stripe.com/docs/payments/checkout
[Accessed 11th April 2022]

Stripe (n.d.) *The Session Object* [online]
Available at: https://stripe.com/docs/api/checkout/sessions/object#checkout_session_object-customer
[Accessed 13th April 2022]

TechWithTim (n.d.) *HTTP Methods* [online]
Available at: https://www.techwithtim.net/tutorials/flask/http-methods-get-post
[Accessed 13th April 2022]

TextMagic (n.d.) *Send an SMS Using Python* [online]
Available at: https://www.textmagic.com/docs/api/python
[Accessed 11th April 2022]

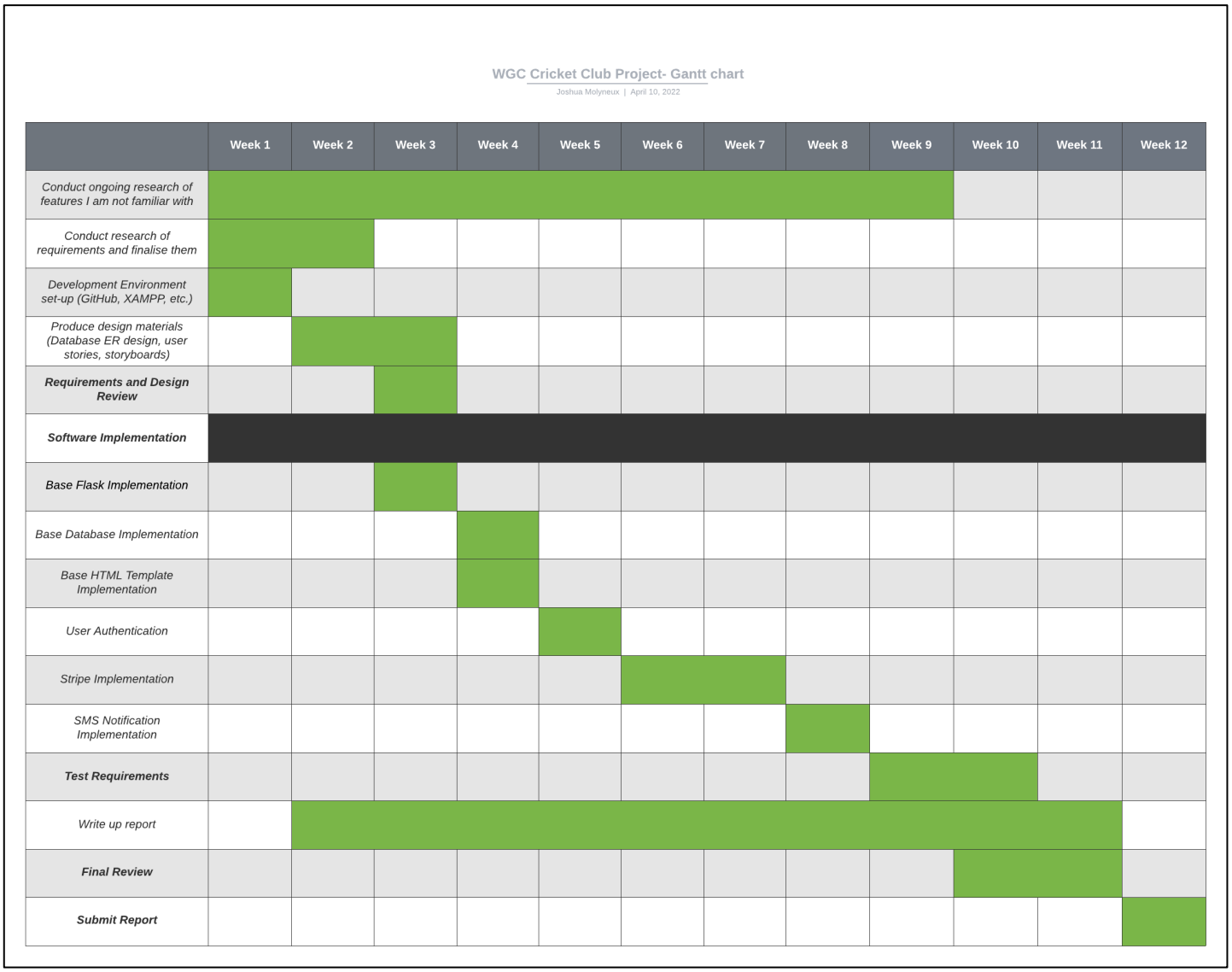
W3Schools (n.d.) *How TO – CSS/JS Modal* [online]
Available at: https://www.w3schools.com/howto/howto_css_modals.asp
[Accessed 11th April 2022]

Watts, S (2020) *ACID Explained: Atomic, Consistent, Isolated & Durable* [online]
Available at: https://www.bmc.com/blogs/acid-atomic-consistent-isolated-durable
[Accessed 4th November 2021]

Welwyn Garden City Cricket Club (n.d.) *New Players* [online]
Available at: https://wgccc.hitssports.com
[Accessed 3rd November 2021]

10.0 Appendices

Appendix A – Gantt Chart for project plan



Appendix B – Verification of Database Table existence in main.py

```
cursor = connect.cursor()
# Create usergroups table if it doesnt already exist
cursor.execute('''
    CREATE TABLE IF NOT EXISTS usergroups(
        group_id INT(6) PRIMARY KEY AUTO_INCREMENT,
        description VARCHAR(255) NOT NULL,
        price_id VARCHAR(100)
    );
''')
cursor.close()
cursor = connect.cursor()
# Create Seniors table if it doesnt already exist
cursor.execute('''
    CREATE TABLE IF NOT EXISTS seniors(
        senior_id INT(6) PRIMARY KEY AUTO_INCREMENT,
        first_name VARCHAR(40) NOT NULL,
        family_name VARCHAR(40) NOT NULL,
        email VARCHAR(100) NOT NULL UNIQUE,
        password VARCHAR(100) NOT NULL,
        address VARCHAR(255) NOT NULL,
        phone_number VARCHAR(30),
        group_id INT(6),
        FOREIGN KEY (group_id) REFERENCES usergroups(group_id)
    );
''')
cursor.close()
cursor = connect.cursor()
# Create Juniors table if it doesnt already exist
cursor.execute('''
    CREATE TABLE IF NOT EXISTS juniors(
        junior_id INT(6) PRIMARY KEY AUTO_INCREMENT,
        first_name VARCHAR(40) NOT NULL,
        family_name VARCHAR(40) NOT NULL,
        dob DATE NOT NULL,
        senior_id INT(6),
        is_developmental BIT(1) DEFAULT 0,
        FOREIGN KEY (senior_id) REFERENCES seniors(senior_id)
    );
''')
cursor.close()
```

```
cursor = connect.cursor()
cursor.execute('''
    CREATE TABLE IF NOT EXISTS subscriptions(
        subscription_id INT(6) PRIMARY KEY AUTO_INCREMENT,
        name VARCHAR(40) NOT NULL,
        description VARCHAR(255)
    );
''')
cursor.close()
cursor = connect.cursor()
cursor.execute('''
    CREATE TABLE IF NOT EXISTS orders(
        order_id INT(6) PRIMARY KEY AUTO_INCREMENT,
        senior_id INT(6),
        subscription_id INT(6),
        order_date DATETIME NOT NULL,
        total_price DECIMAL(5, 2) NOT NULL,
        FOREIGN KEY (senior_id) REFERENCES seniors(senior_id),
        FOREIGN KEY (subscription_id) REFERENCES subscriptions(subscription_id)
    );
''')
cursor.close()
```

Appendix C – Stylesheet for Base HTML Template

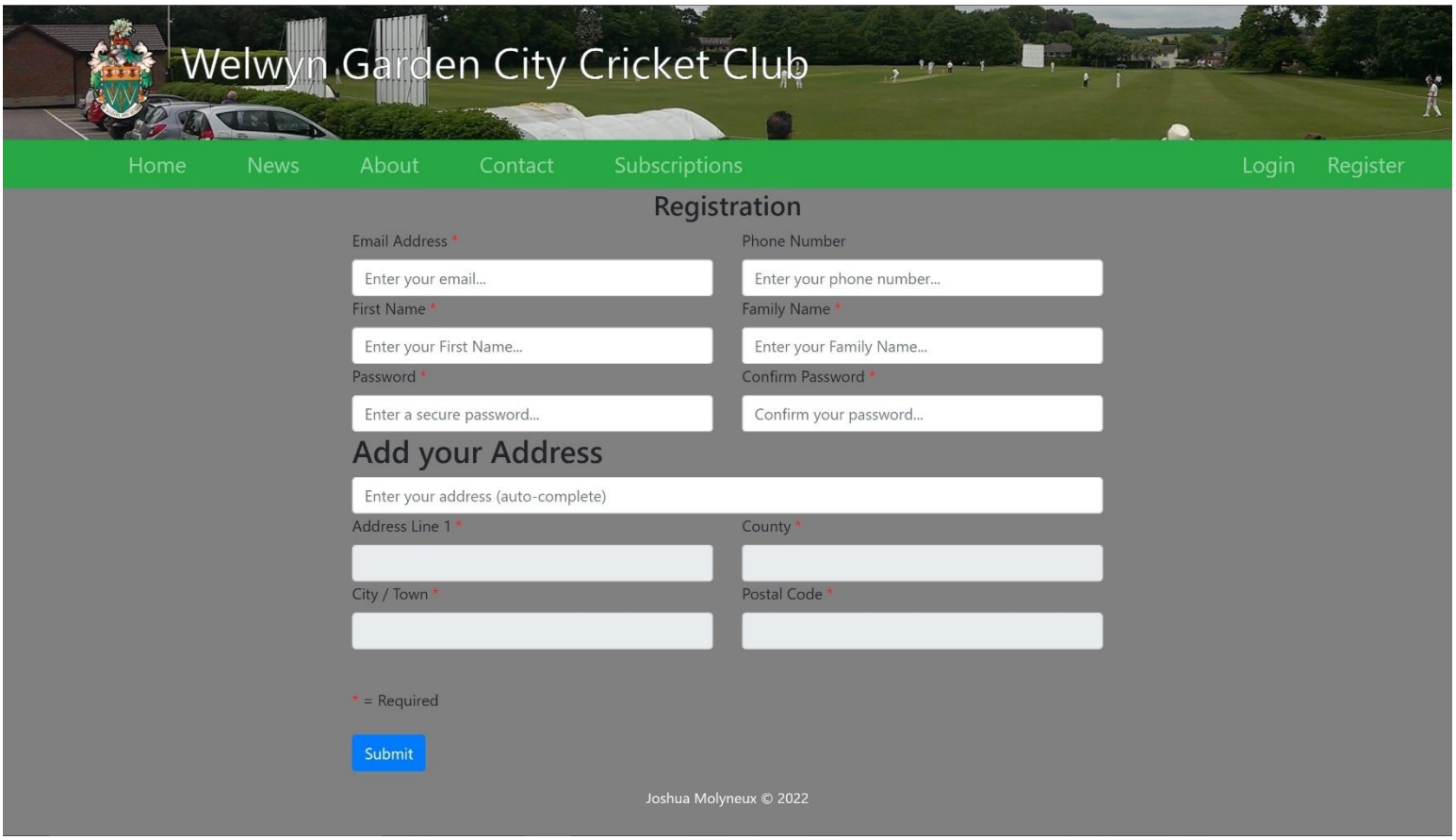
```css
:root {
    --white: #fff;
    --black: #000000;
    --red: red;
    --background: darkgreen;
}

html {
    scroll-behavior: smooth;
    min-width:100%;
}

body {
    background-color: grey
}

#banner {
    background-color: var(--background);
    background-image: url("img/topbanner.jpg");
    background-repeat: no-repeat;
    background-position: center;
    padding-bottom: 10px
}
#banner .banner-text{
    font-size: 3vw;
    text-decoration: none;
    color: var(--white);
    -webkit-text-stroke: 0.5px var(--black);
    margin-inline-start: 5% !important;
}

.logo {
    width: auto;
    height: 15%;
    padding: 10px;
}

.navbar{
    position: sticky !important;
    top: 0 !important;
    padding: 0;
}

.navbar .nav-link.active {
    color: var(--white) !important;
}

.navbar-nav{
    font-size: 1.5vw;
    margin-left: 5% !important;
}

/* THESE NAV ITEMS ARE DEFAULT AS SOON AS YOU ENTER THE WEBSITE*/
.navbar-nav .nav-item.nav-link{
    transition: all 0.4s;
    margin-left: 10% !important;
}

.navbar-nav.ml-auto{
    padding-right: 5%;
}

.navbar-toggler{
    margin-left: 1% !important;
}

@media (max-width: 991.98px) {
    .navbar-nav .nav-item.nav-link {
        margin-left: 5% !important;
        color: var(--white) !important;
        font-size: 3vw !important;
    }

    #banner .banner-text{
        font-size: 4.5vw !important;
    }

    .logo {
        width: auto;
        height: 10%;
        padding: 10px;
    }

    #copyright {
        text-align: center;
        font-size: 2vw;
    }
}

#copyright {
    text-align: center;
    font-size: 1vw;
}
```

Appendix D – Auto-Complete Google Places API integration in JavaScript

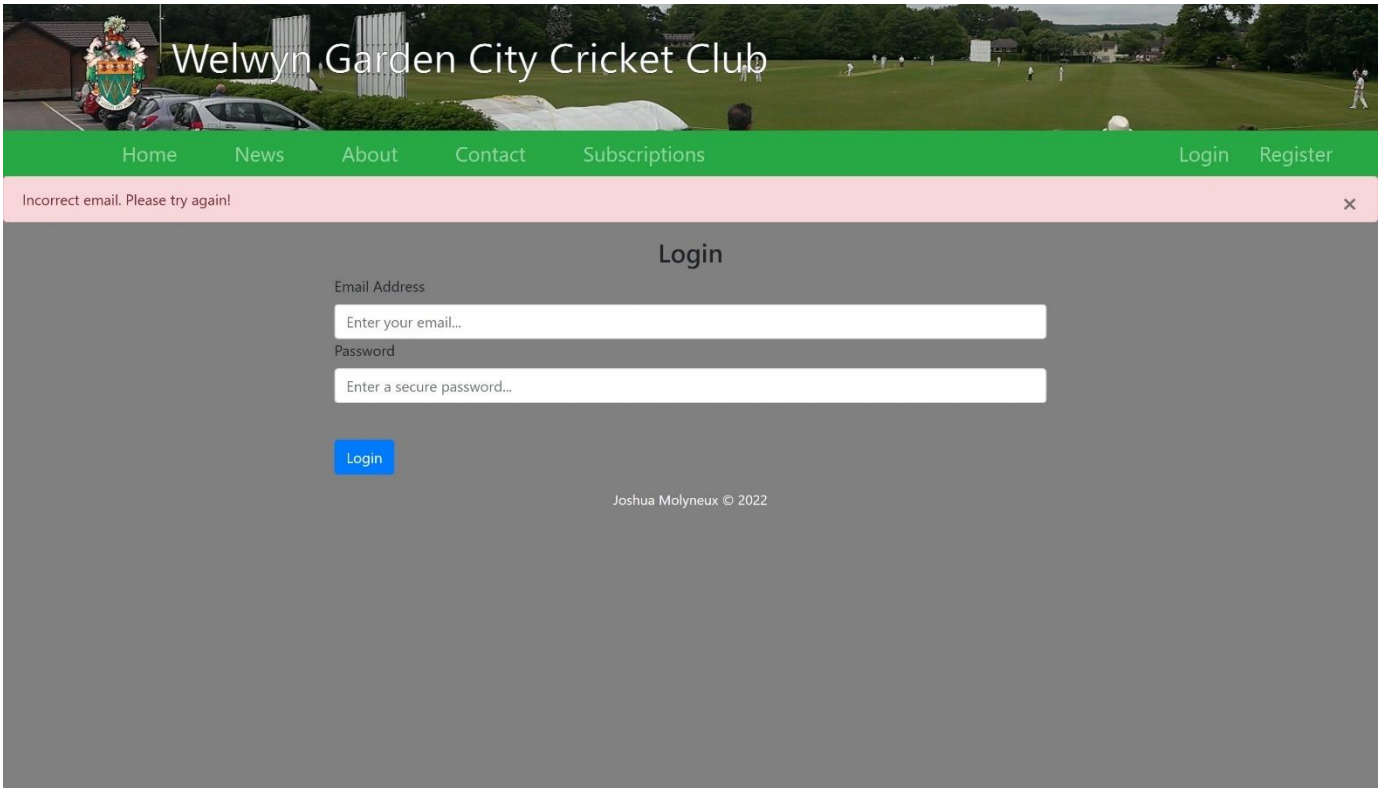
```
// Autocomplete code inspired by: https://stackoverflow.com/questions/31364880/
// combine-street-number-and-route-in-address-autocomplete-api
// Uses Google Places AutoComplete API
var placeSearch, autocomplete;
    var componentForm = {
        street_number: 'short_name',
        route: 'long_name',
        postal_town: 'long_name',
        administrative_area_level_2: 'short_name',
        postal_code: 'short_name'
    };

function initAutocomplete() {
    // Create the autocomplete object, restricting the search to geographical
    // location types.
    autocomplete = new google.maps.places.Autocomplete(
        (document.getElementById('autocomplete')),
        {types: ['geocode']}
    );
    // When the user selects an address from the dropdown, populate the address
    // fields in the form.
    autocomplete.addListener('place_changed', fillInAddress);
}

function fillInAddress() {
    // Get the place details from the autocomplete object.
    var place = autocomplete.getPlace();

    for (var component in componentForm) {
        document.getElementById(component).value = '';
        document.getElementById(component).disabled = false;
    }``
    // Get each component of the address from the place details
    // and fill the corresponding field on the form.
    for (var i = 0; i < place.address_components.length; i++) {
        var addressType = place.address_components[i].types[0];
        if (componentForm[addressType]) {
            var val = place.address_components[i][componentForm[addressType]];
            document.getElementById(addressType).value = val;
        }
    }
    document.getElementById('address1').value =
        place.address_components[0]['long_name'] + ' ' +
        place.address_components[1]['long_name'];
    document.getElementById('address1').disabled = false;
}
```

```
    // Bias the autocomplete object to the user's geographical location,
    // as supplied by the browser's 'navigator.geolocation' object.
    function geolocate() {
        if (navigator.geolocation) {
            navigator.geolocation.getCurrentPosition(function(position) {
                var geolocation = {
                    lat: position.coords.latitude,
                    lng: position.coords.longitude
                };
                var circle = new google.maps.Circle({
                    center: geolocation,
                    radius: position.coords.accuracy
                });
                autocomplete.setBounds(circle.getBounds());
            });
        }
    }
```

Appendix E – Registration webpage



Appendix F: Login webpage (w/ ‘incorrect email’ flash)



Appendix G: Modal implementation in ‘subscriptions.html’

```
<div class='juniors'>
    <div class="modal">
        <div class="modal-content">
            <div class='modal-header'>
                <h2>Junior Subscription</h2>
                <span class="close">&times;</span>
            </div>
            <div class='modal-body'>
                <form name='junior' method="POST">
                    <div class="row">
                        <div class="col-md-6">
                            <label
                                class="control-label"
                                for="junior_first_name">Child's First Name
                                <span class='required'>*</span>
                            </label>
                            <input
                                type="text"
                                class="form-control"
                                id="junior_first_name"
                                name="junior_first_name"
                                placeholder="Enter your Child's First Name..."
                            />
                        </div>
                        <div class="col-md-6">
                            <label
                                class="control-label"
                                for="junior_family_name">Child's Family Name
                                <span class='required'>*</span>
                            </label>
                            <input
                                type="text"
                                class="form-control"
                                id="junior_family_name"
                                name="junior_family_name"
                                placeholder="Enter your Child's Surname..."
                            />
                        </div>
                    </div>
                    <div class='row'>
                        <div class="col-md-6">
                            <label
                                class="control-label"
                                for="junior_dob">Child's Date of Birth
                                <span class='required'>*</span>
                            </label>
                            <input
                                type="date"
                                class="form-control"
                                id="junior_dob"
                                name="junior_dob"
                                />
                            </div>
                        </div>
                        <input type="hidden" id="junior_dev" name="junior_dev" value="junior_dev">
                        <button id='checkout' class="btn btn-primary" type="submit">Checkout</button>
                    </form>
                </div>
            </div>
        </div>
        <h1>Junior Subscriptions</h1>
        <div class="product-container">
            <ul class='product-card'>
                <li class='name'>Junior Member</li>
                <li class='short_desc'>*social membership included</li>
                <li class='price'>£135.00</li>
                <li><button class="btn btn-primary openmodal">Options</button></li>
            </ul>
            <ul class='product-card'>
                <li class='name'>Junior Member (up to 7/8 years old)</li>
                <li class='short_desc'>*social membership included</li>
                <li class='price'>£80.00</li>
                <li><button class="btn btn-primary openmodal">Options</button></li>
            </ul>
        </div>
    </div>

{%endblock%}
```

Appendix H – Directory Structure