MMA Fantasy League Project
Development and Testing Blog

Members:
Paul Bashford: 19709791
James Anthony Reilly: 19464192

Supervisor: Mark Roantree

Date Completed: 24/02/2023

Table of Contents

1. Foreword
   1.1 Introduction
2. Backend
   2.1 Sign Up/In
   2.2 Fighter Data
   2.3 Creating Leagues
   2.4 Join League
   2.5 League Table

3. Frontend
   3.1 Base HTML Structure
   3.2 CSS and Javascript
   3.3 Homepage
   3.4 Sign In
   3.5 Sign Up
   3.6 Create/Join League
   3.7 Choose Fighters
   3.8 Selected Fighters
   3.9 League Table
4. Unit Tests
   4.1 Testing Methods
   4.2 Sign In Test Case
   4.3 Choose Fighters and Save Selection
   4.4 Create League
   4.5 Join League
   4.6 League Table
5. User Testing
   5.1 Mock UFC Event Test
   5.2 Changes After User Feedback

Foreword

1.1 Introduction

Before starting any coding for our project we decided that we needed to lay out a step by step plan of how we were going to develop the project and the particular steps we were going to take. We decided that the steps we were going to take for this were as follows:

1. Create the Django Project and file layout.
2. Design the Django Models to design the database structure for the MMA fantasy league. This would include models for fighters, teams, matches, point system, league and transactions.
3. Integrate Firebase. This would be ideally done using the pyrebase library to connect django to firebase
4. Create views and URLs. This would be done by using django views to render templates and process form resubmissions. We will have to create URLs for each view and map them to the corresponding view function
5. Design the Templates. We will use Django to design the user interface for the webapp. We will make use for HTML, CSS and Javascript
6. Deploy to Firebase. Begin hosting of our application on the firebase server.
7. App Testing. We will then test the web app to ensure it works as expected and then get to work designing our ethical approval forms and give the app to some of our peers for a broader scale of user testing.

Backend

2.1 Sign Up/In

To start off we made a simple signup function that takes an argument and the server will respond by rendering the "signUp.html" template and sending it back as the HTTP response to the client's browser.

```
def signUp(request):

  return render(request, "signUp.html")
```

This function was working fine in loading up the relevant html page but we needed to write another function that would actually take information from the user and send and store this data to the database so it could be used as their login parameters. To do this we created the **postsignup** function.

```
def postsignup(request):

 name=request.POST.get('name')
 email=request.POST.get('email')
 passw=request.POST.get('pass')


 try:
   user=authe.create_user_with_email_and_password(email,passw)
   uid = user['localId']
   data={"name":name,"email":email,"status":"1"}
   database.child("users").child(name).child("details").set(data)
   request.session['name'] = name
   request.session['email'] = email
   request.session.save()


 except:
   message="Unable to create account try again"
   return render(request,"signup.html",{"messg":message})

 return render(request,"signIn.html")
```

This function works by taking a "request" object as an argument. The "request" object contains information about the current HTTP request made by a client to the web server.

The function retrieves the user's name, email, and password from the form data submitted through the POST request. It then attempts to create a new user account using the email and password provided by the user, using a Firebase Authentication API call to create_user_with_email_and_password.

If the user account creation is successful, the function saves the user's name, email, and a status value of "1" (indicating that the user account is active) in the Firebase Realtime Database under the "users" node with the user's name as the key.

The function then creates a new session for the user and stores their name and email in the session variables. Finally, it renders a "signIn.html" template to direct the user to the sign-in page to log in to their new account.

If the user account creation is not successful for any reason, the function catches the exception and returns a "signup.html" template with an error message indicating that the account could not be created.

```
uid = None
```

We had some problems with the log in system and the post sign up page but after returning after doing some research we realised that our uid variable had to be defined before the try statement and we also had to sync up the firebase database url with our config key. To do this we set the uid equal to None

After this we realised that our ‘uid’ wasn’t updating to our realtime database on the firebase tool. We were stuck with this problem for quite some time but realised that our previous fix by setting the ‘uid’ to “None” was preventing the database from being updated and thus not being set to the correct value if the user creation was successful. We fixed this by integrating the ‘uid’ assignment inside our try statement.

```
try:
  user=authe.create_user_with_email_and_password(email,passw)
  uid = user['localId']
  data={"name":name,"email":email,"status":"1"}
  database.child("users").child(name).child("details").set(data)
  request.session['name'] = name
  request.session['email'] = email
  request.session.save()
```

2.2 Fighter data

Next we moved onto designing the models for use with the database. To start off we made a simple fighter class with variables for the fighter’s name, record etc. then we made a save function to store this data to the database. We messed around with ideas on how to send the data to the database most efficiently and after a while we landed on the method of having one python script that contained a dictionary with each fighter’s details such as their names, ages, records, weight classes, and points.

```
import pyrebase

firebaseConfig = {
  'apiKey': "AIzaSyD-_zfDFYhcZUDpTTaKS7I0YJahMaqiCLs",
  'authDomain': "mma-fantasy-league-94b6e.firebaseapp.com",
  'databaseURL':
"https://mma-fantasy-league-94b6e-default-rtdb.firebaseio.com",
  'projectId': "mma-fantasy-league-94b6e",
  'storageBucket': "mma-fantasy-league-94b6e.appspot.com",
  'messagingSenderId': "837434946169",
  'appId': "1:837434946169:web:7a0210a7c523225e629aa8",
  'measurementId': "G-DRKKN38R7H"
}
```

The code imports the pyrebase library and initializes a Firebase app using a dictionary of configuration settings that define the Firebase project's API key, authentication domain, database URL, project ID, storage bucket, messaging sender ID, and app ID.

```
firebase=pyrebase.initialize_app(firebaseConfig)
db = firebase.database()
data = {
        'name': 'Jessica Andradre',
        'age': 31,
        'record': '24-9-0',
        'weight_class': 'Flyweight',
        'points':3
    }
db.child("Fight cards").child("UFC FIGHT NIGHT: Andradre v Blanchfield
").child("Main Card").child("Fights").child("Andradre v
Blanchfield").child("Jessica Andradre").set(data)
```

The code uses the Firebase Realtime Database to create a hierarchical data structure containing information about each fighter and their respective fight cards. Specifically, it creates a tree structure under the "Fight cards" node, with each fight card having its own child node. The Main Card node has its own child node for each fight, with each fighter having its own child node that contains their respective data.

The code uses the set() method of the database reference to add the data to the database, overwriting any existing data at that location if there is any.

2.3 Creating Leagues

Next we moved onto the functions for creating leagues. The first function **createleague(request):** takes a request object as input, which contains information about a web request made by a user. The function then returns the corresponding html page for after a league has been created.

```
def createleague(request):
    email = request.session.get("email")
    return render(request,"createleague.html",{"email": email})
```

The second function **postcreateleague(request):** also takes a request object as input. It first checks if the user is logged in by looking for a 'uid' value in the session object. If the user is logged in, it retrieves information from the request object about a league that the user is trying to create, such as the league name and a unique code to identify the league. The function then creates a dictionary with this information and some additional information about the league, such as the owner's email and a list of league members that initially only contains the owner.

```
def postcreateleague(request):
  if 'uid' in request.session:
    id_token = request.session['uid']
    try:
        leaguename = request.POST.get('leaguename')
        uniquecode = request.POST.get('uniquecode')
        owner = request.session.get("email")
        members = [owner]
        data = {"leaguename": leaguename, "owner": owner, "members":
members, "uniquecode": uniquecode}
        database.child("leagues").child(leaguename).set(data)
        print(uniquecode)
        print(leaguename)
        return redirect('choosefighters',)
    except:
        message="Unable to create account try again"
        return render(request,"signup.html",{"messg":message})
  else:
        message="You are logged out, to continue log back in!"
        return render(request,"signIn.html", {"messg":message})
```

The function then pushes this league dictionary to a database using the Firebase API. If the push is successful, the function prints the unique code and league name to the console and redirects the user to another web page called "choosefighters". If there is an error with the push operation, the function catches the error and renders "signup.html" with an error message.

If the user is not logged in, the function renders "signIn.html" with a message prompting the user to log in before continuing.

2.4 Join League

Following this we moved onto the **joinleague** functionality. To do this we needed to create two separate functions once again. The first function works virtually the same as the function for **createleague**.

```
def joinleague(request):
    email = request.session.get("email")
    return render(request,"joinleague.html",{"email": email})
```

The second function works by taking a request object as input, which contains information about a web request made by a user. It first checks if the user is logged in by looking for a 'uid' value in the session object. If the user is logged in, it retrieves the user's email from the session object and the unique code entered by the user on the web page.

```
 if 'uid' in request.session:
```

```
        id_token = request.session['uid']
```

The function then gets a dictionary of all the leagues from a database using the Firebase API. It sets a boolean variable **league_found** to False and then iterates over the dictionary to find a league that matches the unique code entered by the user. If the unique code matches, the function checks if the user is already a member of the league. If the user is already a member, it renders "joinleague.html" with a message informing the user that they are already a member of the league.

```
try:
            uniquecode = request.POST.get('uniquecode')
            email = request.session.get("email")
            leagues = database.child("leagues").get().val()
            league_found = False
            for key, league in leagues.items():
                print("League key:", key)
                print("League unique code:", league['uniquecode'])
                print("League members:", league['members'])
```

If the user is not already a member, the function adds the user's email to the list of league members and updates the database with this new information. It then sets league_found to True and breaks out of the loop. If no league is found with the entered unique code, the function renders "joinleague.html" with a message informing the user that the unique code they entered does not match any existing leagues.

```
                if league['uniquecode'] == uniquecode:
                    if email in league['members']:
                        message = "You are already a member of this
league."
                        return render(request, "joinleague.html",
{"messg": message})
                    else:
                        members = league['members']
                        members.append(email)

database.child("leagues").child(key).update({"members": members})
                        league_found = True
                        break
            if league_found:
                print("League found. Redirecting to leaguetable.")
                return redirect('leaguetable')
            if not league_found:
                message="The unique code you entered does not match any
existing leagues. Please try again or create a new league."
```

```
                return render(request, "joinleague.html",
{"messg":message})
```

If there is an error with the database operations or the unique code entered by the user, the function catches the error and renders "joinleague.html" with an error message.

```
        except Exception as e:
            print("Exception:", e)
            message="The unique code you entered does not match any
existing leagues. Please try again or create a new league."
            return render(request, "joinleague.html",
{"messg":message})
    else:
        message="You are logged out, please log back in to join a
league."
        return render(request, "signIn.html", {"messg":message})
```

If the user is not logged in, the function renders "signIn.html" with a message prompting the user to log in before continuing. Finally, if everything is successful, the function redirects the user to the league table.

2.5 League Table

Finally we moved onto the **leaguetable** function. This works by first checking if the user is logged in by looking for a 'uid' value in the session object. If the user is logged in, it retrieves the user's email from the session object.

```
try:
            leagues = database.child("leagues").get().val()
            leagues_dict = dict(leagues)
            data = []
            for league_id, league in leagues_dict.items():
                if 'members' in league and email in league['members']:
                    print('User is a member of league:',
league['leaguename'])
                    league_data = {'name': league['leaguename'],
'members': []}
                    for member in league['members']:
                        if member is None:
                            continue
                        print('Processing member:', member)
                        member_encoded_email = member.replace('.', ',')
```

```
                        points =
database.child("users").child(member_encoded_email).child("points").chi
ld("total_points").get().val()
                        if points is None:
                            points = 0
                        member_data = {'name': member, 'points':
points}
                        print(member_data)
                        league_data['members'].append(member_data)
                    league_data['members'] =
sorted(league_data['members'], key=lambda x: x['points'], reverse=True)

                    data.append(league_data)
            if not data:
                message="You are not part of any leagues, join or
create a league."
                return render(request, "welcome.html",
{"messg":message})
return render(request, 'leaguetable.html', {'data': data})
```

The function then gets a dictionary of all the leagues from a database using the Firebase API. It converts the dictionary to a regular Python dictionary and then creates an empty list called data. It iterates over each league in the dictionary and checks if the user is a member of the league by looking for the user's email in the members list of the league. If the user is a member of the league, the function creates a dictionary called league_data that contains the league name and a list of all the members of the league.

The function then appends this league_data dictionary to the data list. If the data list is empty, the function renders "welcome.html" with a message informing the user that they are not part of any leagues and prompting them to create or join a league.

If the data list is not empty, the function renders "leaguetable.html" and passes the data list as a parameter. This page displays a table of all the leagues that the user is a member of, along with an ordered list of all the members in each league.

If there is an error with the database operations, the function catches the error and renders "joinleague.html" with an error message.

If the user is not logged in, the function renders a "signIn.html" with a message prompting the user to log in before continuing.

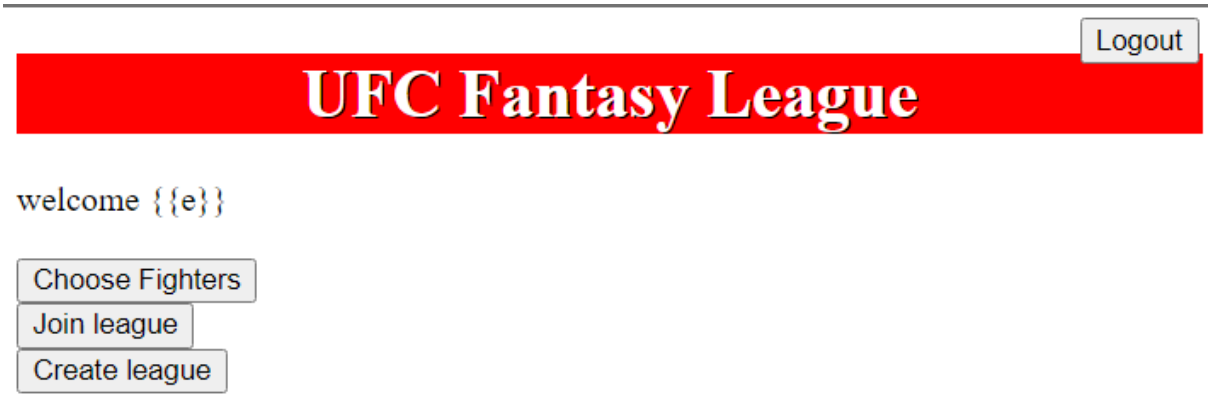
Frontend

3.1 Base Html Structure

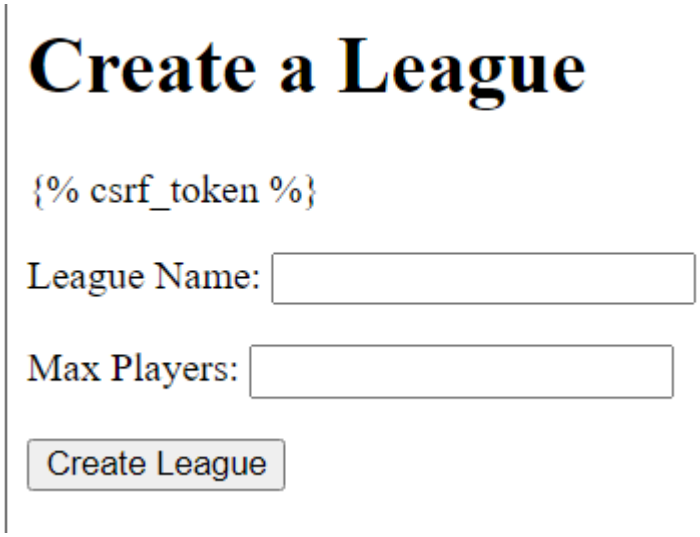
When starting our front end development we decided to design some very basic HTML pages for each intended function that would give us an idea of the application's layout with the intention of adding CSS and some java at a later date to make the pages look more aesthetically pleasing.

A few examples of these simple html pages are shown below.

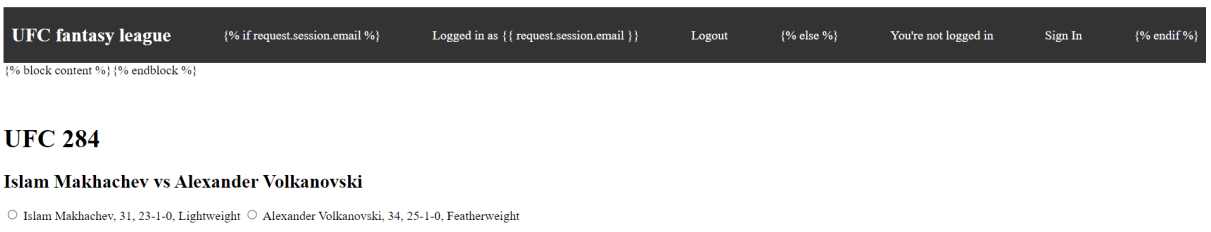
Welcome.html



Createleague.html



Choosefighters.html



3.2 CSS and Javascript

The first step to making our html pages more aesthetically pleasing was to incorporate some css, javascript and of course images into the files. To do this we set up a "static" folder in our

project directory that contained two separate folders that the html files would pull the relevant images and css files from.

Once we had this done the next step was to simply add the following code to make our pages access the relevant css code

```
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>MMA Fantasy League</title>
    <link rel="stylesheet" type="text/css" href="{% static
'css/header.css' %}">
    <link rel="stylesheet" type="text/css" href="{% static
'css/chFighter.css'%}">
</head>
```

Images were also similarly integrated with an example shown below

```
<div class="logo-container">
                    <img src="{% static 'images/logo.png' %}"
alt="log">
                </div>
```

3.3 Homepage

The Homepage of our application was the foundation of the frontend aspect of the project in that every other page borrowed quite a bit of code from it as most of the html pages are quite similar save for some special functions. Most of the pages also share a background image of a MMA cage.

First off we made the header section that would be the same for all the pages except sign up and sign in. The header consists of two sections, the left and right section. The left displays the website logo and the title of the application. The right section contains a navigation bar with two options, "Logout" and "Sign In", depending on whether the user is currently logged in or not. If the user is logged in, their email address is also displayed. Additionally, there is a "Home" button that takes the user to the homepage of the website. The classes assigned to the different HTML elements allow for CSS styling to be applied to create a visually appealing header section.

```
 <div class="header">
        <div class="container">
            <div class="left">
                <div class="logo-container">
                  <img src="{% static 'images/logo.png' %}" alt="log">

                </div>
                <h1>UFC fantasy league</h1>
```

```
            </div>
            <div class="right">
                <nav>
                    {% if request.session.email %}
                    <p>Logged in as {{ request.session.email }}</p>
                    <a href="{% url 'logout' %}">Logout</a> {% else %}
                    <p>You're not logged in</p>
                    <a href="{% url 'signIn' %}">Sign In</a> {% endif
%}
                </nav>
                <a class="get-start-btn" href="home">Home</a>
            </div>
        </div>
    </div>
```

Next we moved on to the main section of the page. This code creates the main content of the webpage, which includes a welcome message and a set of buttons that allow users to navigate to different pages within the site. The welcome message is displayed within a container that has a class of "content". This container is also used to define content blocks that can be replaced by child templates.

```
<main>
        <div class="container">
            <div class="content">
                Step into the Octagon: Welcome to the Ultimate
<br><span class="bold-text">UFC Fantasy League</span> {% block content
%}{% endblock %}
            </div>
            <br><br>
            <div class="btn-container">

              <button type="button" onclick="location.href='{% url
'fighter_selection' %}'">Selected Fighters</button><br>

              <button type="button" onclick="location.href='{% url
'choosefighters' %}'">Choose fighters</button><br>

              <button type="button" onclick="location.href='{% url
'joinleague' %}'">Join league</button><br>
              <button type="button" onclick="location.href='{% url
'createleague' %}'">Create league</button><br>
              <button type="button" onclick="location.href='{% url
'leaguetable' %}'">League Table</button>
            </div>
```

```
        </div>
    </main>
    <script src="{% static 'js.js' %}"></script>
</body>
```

The buttons are defined as a group of HTML buttons that are wrapped in a container that has a class of "btn-container". Each button has an "onclick" attribute that defines the URL to navigate to when the button is clicked. These URLs are defined using Django's URL routing system, which is referenced using template tags in the "onclick" attribute. The last line of the code includes a reference to a JavaScript file, "js.js", which contains custom JavaScript code that is used to enhance the functionality of the buttons.

3.4 Sign In

The sign in page is quite simple in that it also contains the header but there are no buttons on the right side to navigate to any other pages.

The main section of the pages contains a main element that wraps the form, which is enclosed in a div with a class of "log-box". The left side of the div contains a container with two input fields, one for the email and the other for the password. The form has a POST action that points to a URL called "postsign". It also includes a CSRF token to protect against cross-site request forgery attacks.

```
   <main>
        <div class="log-box">
            <div class="left">
                <div class="inner-left-container">
                    <div class="content">
                        Sign In {% block content %}{% endblock %}
                    </div>
                    <br><br>
                    <form action="postsign" method="post">
                        {% csrf_token %}
                        <div class="inp-container">
                            <input type="email" name="email"
placeholder="Email">
                            <input type="password" name="pass"
placeholder="Password">
                        </div>
                        <div class="inp-buttons">
                            <input type="submit" value="Sign In">
                            <button type="button"
onclick="location.href='{% url 'signUp' %}'">Sign up</button>

                        </div>
                    </form>
```

```
                </div>
            </div>
            <div class="right d-flex al-center">
                <img src="{% static 'images/player-vector.png' %}"
alt="player">
        </div>
    </main>
```

The login form has two buttons: "Sign In" and "Sign up". The "Sign In" button will submit the form to the server, while the "Sign up" button will redirect the user to the "signUp" URL when clicked. Finally there is an image loaded in that is unique to this page.

3.5 Sign Up

The sign up page is virtually identical to the sign in page only that the main div containing the central box includes a form with input fields for name, email, and password, along with a "Sign up" button.

```
 <div class="inp-container">
                            <input type="text" name="name"
placeholder="Name">
                            <input type="email" name="email"
placeholder="Email">
                            <input type="password" name="pass"
placeholder="Password">
                        </div>
                        <div class="inp-buttons">
                            <input type="submit" value="Sign up">
                        </div>
```

The user's input is posted to the database when the sign up button is clicked.

3.6 Create league/Join League

The create and join league are also quite similar to the login page. Both pages share the header similar to the other pages and also have extremely similar <main> sections.

The only difference being that the create league prompts the user to input a league name and uniquecode which other players use to join the league. The data is added to the database when the create button is clicked.

```
                    <input type="text" name="leaguename"
placeholder="League Name"><br>
                     <input type="text" name="uniquecode"
placeholder="Uniquecode"><br>
                    <input type="submit" value="Create">
```

The join league page only allows users to input a unique code and upon clicking the join button it checks the database to see if the input value matches that of any league and if it does the user is added to the league.

```
                <form action="postjoinleague" method="post">
                    {% csrf_token %}
                    <input type="text" name="uniquecode"
placeholder="uniquecode"><br>
                    <input type="submit" value="Join">
                </form>
```

3.7 Choose Fighters

The choose fighters page displays two different sections and a form that allows users to select their preferred fighters for two matchups on the fight card. The form has radio buttons for each matchup that allow users to select their preferred fighter. . One section is for the main event fights and the other is for the preliminary fights. These boxes have all of the fights scheduled to take place at the next UFC event. This is achieved by having different classes to handle all of the relevant information. The “box” class is used to display a section for each relevant fight and allow users to make their choice of fighter. An example of this is shown below.

```
                            <div class="box">
                                <h2>Andradre v Blanchfield</h2>
                                <ul>
                                    <li>
                                        <input type="radio"
name="andradre-blanchfield" value="Jessica Andradre">
                                        <label for="Jessica
Andradre">Jessica Andradre</label>
                                    </li>
                                    <li>
                                        <input type="radio"
name="andradre-blanchfield" value="Erin Blanchfield">
                                        <label for="Erin
Blanchfield">Erin Blanchfield</label>
                                    </li>
                                </ul>
                            </div>
```

We also used a container class to encapsulate all of the boxes and map them to either the main or preliminary events.

3.8 Selected Fighters

The selected fighters page differs depending on whether the user has made their selections already.

If a selection has been made then the code will display a table with two columns: "Fight" and "Fighter Selection".

The table is constructed using an HTML table element with the class "my-table". It has a header row with two cells for the column headers, "Fight" and "Fighter Selection".

The table body is constructed using a for loop that iterates over the fighters_selected dictionary. For each key-value pair in the dictionary, the loop creates a new row in the table with two cells: one for the fight name and one for the selected fighter's name.

Finally, the code includes a link with the text "CLICK HERE TO CHANGE FIGHTERS" that points to the URL "/choosefighters".

If the user has not made their selection then the page displays a string stating that no selection has been made and displays a button that directs the user to the choosefighters page.

3.9 League Table
The league table page generates a table displaying the rankings and points for a given league and its members.

```
  <div class="t-center">
   <h1>The {{ league.name }} league</h1>
   </div>
<table class="my-table">
  <thead>
    <tr>
      <th>Rank</th>
      <th>User</th>
      <th>Points</th>
    </tr>
  </thead>
  <tbody>
    {% for member in league.members %}
      <tr>
        <td>{{ forloop.counter }}</td>
        <td>{{ member.name }}</td>
        {% if member%}
          <td>{{ member.points }}</a></td>
        {% else %}
          <td>No Selections Made</td>
        {% endif %}
      </tr>
    {% endfor %}
```

```
    </tbody>
  </table>
  <br><br>
{% endfor %}
```

The first line of code displays the league name with {{ league.name }}.
The next lines create a table with three columns: Rank, User, and Points.
The for loop iterates through each member in the league, and for each member, the code displays the rank, name, and points in the respective columns.
If a member has not made any selections yet, the code displays "No Selections Made" in the Points column.

Unit Tests

4.1 Testing Methods

We had of course been manually testing all of our functionality as we were designing it but we decided it would be more efficient to write some test scripts to ease the process of testing and to easily collect data from our tests. We decided to use the magicmock library to help with our testing as it allowed us to provide a simple mocking interface that allowed us to set the return value or other behaviour of the functions. This allowed us to fully define the behaviour of the call and avoid creating real objects. This was extremely useful for the testing process as we didn't have to go to the database and delete any data after any failed tests. All script written tests were executed by python files that either output the results to the terminal to show us the results of the tests or executed without an error message indicating that the data sent to the database correlated with the data that was received from the database.

4.2 Sign in Test Case

We started off by writing a script to test the sign in functionality. We started by creating a class for our post sign in testing and then defining two functions to test a user login with valid and invalid credentials.

The code sets up a Django environment and runs two test cases using the Django TestCase framework. The test cases are testing the postsign view function in the backend Django app. The "@patch" decorator is used to replace the Firebase Authentication API call to sign_in_with_email_and_password

The first test case (test_postsign_valid_credentials()) tests the scenario where the user enters valid credentials (email and password). The test creates a mock request with the valid email and password, calls the postsign function with the mock request, and verifies that the response status code is 200, the welcome.html template is used, and the session variables for email and uid are set correctly.

The second test case (test_postsign_invalid_credentials()) tests the scenario where the user enters invalid credentials (incorrect email or password). The test creates a mock request

with the invalid email and password, calls the postsign function with the mock request, and verifies that the response status code is 200, the signIn.html template is used, and the error message for invalid credentials is displayed.

When the code is executed, the test results will be printed to the console, indicating whether the tests passed or failed. The output will include information about the test cases that ran, any errors that occurred, and the status of each test (passed or failed).

```
class PostSignTestCase(TestCase):

    @patch('backend.views.authe.sign_in_with_email_and_password')
    def test_valid_credentials(self, mock_sign_in):
        mock_sign_in.return_value = {'idToken': '12345'}
        request = RequestFactory().post('/', {'email':
'test@example.com', 'pass': 'password'})
        response = postsign(request)
        self.assertEqual(response.status_code, 200)
        self.assertContains(response, 'test@example.com')

    @patch('backend.views.authe.sign_in_with_email_and_password')
    def test_invalid_credentials(self, mock_sign_in):
        mock_sign_in.side_effect = Exception('Invalid credentials')
        request = RequestFactory().post('/', {'email':
'test@example.com', 'pass': 'password'})
        response = postsign(request)
        self.assertEqual(response.status_code, 200)
        self.assertContains(response, 'Invalid Credentials')
```

The script ran flawlessly as expected but it was nice to have some easily recordable and displayable data.

```
C:\Users\Paul\Desktop\testing>python postsignt.py
System check identified no issues (0 silenced).
..
----------------------------------------------------------------------
Ran 2 tests in 0.003s

OK
```

4.3 Choose Fighters and save Selection

Next we moved onto the testing of our fighter selection functionality. The choose fighter function works by loading a html page that allows users to input their selection for the upcoming event. The user selection is recorded by a form that is filled out on the html page and when the user clicks the submit button the **save_choices** function runs and posts their selections to the database. We had quite some trouble with getting this function to work so most of the testing for this was done by manually inputting the selections and checking the

database to see if the selections were being saved correctly. (Selections saved to database in image below).

After we had got the manual functionality working we went onto writing a similar script that we could run to test the functionality more easily.

```
class SaveChoicesViewTest(TestCase):

    @patch('backend.views.database.child')
    def test_save_choices_view(self, mock_child):
        session_data = {
            'uid': 123,
            'name': 'John Doe',
            'email': 'john.doe@example.com',
        }
        self.client.session.update(session_data)
        self.client.session.save()
```

Once again the `@patch()` segment of the code is used to create a mock object that allows us to test without actually changing anything. The method then creates a dictionary **session_data** to simulate a user session with some user data. Then, it updates the session object of the Django test client with the session data and saves it. This simulates a user being logged in to the system and ready to save their fighter choices.

```
url = reverse('save_choices')
        data = {
            FIGHTER DATA HERE,
        }
        response = self.client.post(url, data)

        encoded_email = session_data['email'].replace('.', ',')
        mock_child.return_value.set.assert_called_once_with({
            FIGHTER DATA HERE
        })
```

This code is making a POST request to a specific URL with a set of data containing fighter choices. The server should receive this request and save the fighter choices data to a database.

Then, the code is checking if the fighter choices data has been properly saved to the database by using a mock object to check if the set method of the database child is called once with the correct data. If the set method is called with the correct data, then it means the fighter choices have been successfully saved to the database.

4.4 Create League

Our create league functionality worked similarly in that it used two functions once again: a function for loading up the html page and a function to post the league data to the database.

```
    @patch('myapp.views.database.child')
    def test_post_create_league_success(self, mock_child):
        request = RequestFactory().post('/createleague', {'leaguename':
'My League', 'uniquecode': 'ABC123'})
        request.session = {'uid': 'user_id', 'email':
'user@example.com'}
        request.session.save()
        response = postcreateleague(request)
        self.assertEqual(response.status_code, 302)
        self.assertEqual(response.url, '/choosefighters')
        mock_child.assert_called_once_with('leagues/My League')
        mock_child.return_value.set.assert_called_once_with({
            'leaguename': 'My League',
            'owner': 'user@example.com',
            'members': ['user@example.com'],
            'uniquecode': 'ABC123'
        })
```

To test the post function we made three different test cases. For the first test case we simulate a successful request where the user is logged in and provides valid data. We use patch to mock the database.child function and assert that it is called with the correct arguments. We also assert that the set method of the mocked child is called with the expected data, and that the response is a redirect to the choosefighters page.

```
    def test_post_create_league_not_logged_in(self):
        request = RequestFactory().post('/createleague', {'leaguename':
'My League', 'uniquecode': 'ABC123'})
        request.session = {}
```

```
        response = postcreateleague(request)
        self.assertEqual(response.status_code, 200)
        self.assertContains(response, 'You are logged out, to continue
log back in!')
        self.assertTemplateUsed(response, 'signIn.html')
```

In the second test case, we simulate a request where the user is not logged in. We assert that the response is a render of the signIn.html template with the expected error message.

```
    def test_post_create_league_error(self):
        request = RequestFactory().post('/createleague')
        request.session = {'uid': 'user_id', 'email':
'user@example.com'}
        request.session.save()
        with patch('myapp.views.database.child') as mock_child:
            mock_child.return_value.set.side_effect =
Exception('Database error')
            response = postcreateleague(request)
        self.assertEqual(response.status_code, 200)
        self.assertContains(response, 'Unable to create account try
again')
        self.assertTemplateUsed(response, 'signup.html')
```

In the third test case, we simulate a request where there is an error in the database operation. We use patch again to mock the set method of the child and make it raise an exception. We assert that the response is a render of the signup.html template with the expected error message.

4.5 Join league

The testing of the Join League functionality was fairly straightforward. The purpose of this test is to ensure that the function works correctly and can join a user to a league in the database. It consisted of a script that created a mock database response and mock request object to simulate the behaviour of the function.

```
class JoinLeagueTest(TestCase):
    def setUp(self):
        self.factory = RequestFactory()

    @mock.patch('yourapp.views.database')
    def test_postjoinleague(self, mock_db):
        mock_leagues = {
            'league1': {'uniquecode': '123', 'members':
['user1@example.com']},
```

```
            'league2': {'uniquecode': '456', 'members':
['user2@example.com']},
        }
        mock_db.child.return_value.get.return_value.val.return_value =
mock_leagues
        session = {'uid': 'someuid', 'email': 'user1@example.com'}
        post_data = {'uniquecode': '123'}
        url = reverse('postjoinleague')
        request = self.factory.post(url, post_data)
        request.session = session
        response = postjoinleague(request)
        self.assertRedirects(response, reverse('leaguetable'))
        mock_db.child.return_value.child.assert_called_with('league1')

mock_db.child.return_value.child.return_value.update.assert_called_once
_with({'members': ['user1@example.com',]})
```

It then calls the function with the mock request object and checks if the response is a redirect to the league table page. It also checks if the database update function was called with the correct parameters.

4.6 League table

To test our league table's performance we designed a test for a logged in user and an unauthenticated user.

```
class LeagueTableTest(TestCase):
    def setUp(self):
        self.factory = RequestFactory()

    @patch('yourapp.views.database')
    def test_league_table_authenticated_user(self, mock_db):
        session = {'uid': 'someuid', 'email': 'test@example.com'}
        mock_leagues = {
            'league1': {'leaguename': 'League 1', 'members':
['test@example.com', 'other@example.com']},
            'league2': {'leaguename': 'League 2', 'members':
['another@example.com']},
        }
        mock_db.child.return_value.get.return_value.val.return_value =
mock_leagues
        url = reverse('leaguetable')
        request = self.factory.get(url)
        request.session = session
```

```
        response = leaguetable(request)
        self.assertContains(response, 'League 1')
        self.assertContains(response, 'test@example.com')
        self.assertContains(response, 'other@example.com')
        self.assertNotContains(response, 'another@example.com')

    def test_league_table_unauthenticated_user(self):
        url = reverse('leaguetable')
        request = self.factory.get(url)
        response = leaguetable(request)
        self.assertContains(response, 'You are logged out, to continue
log back in!')
```

using a mock database response to simulate the data that the function would normally retrieve from the database. It checks that the response contains the expected league and member data, or the expected error message if the user is not authenticated.

User Testing

5.1 Mock UFC Event test

For our user acceptance testing we approached a number of our peers who were interested in MMA and gave them access to the application to test it. Due to the fact that we didn't manage to get our functionality working for the UFC event, we had to simulate a UFC event with randomly selected winning fighters and get our friends to make a selection of their team of fighters. After they had made their selection and seen their results posted to the league table, we asked them to fill out a form with a number of questions for feedback on the application.

5.2 Changes after user feedback

A number of our testers stated on the feedback section of the form that they didn't like the fact that they were restricted to only being able to choose five fighters for each event and that they would prefer if they could choose a winner from each fight taking place on the night.

Is there anything that you think could be improved in the application?

4 responses

I didn't like how I was restricted to only choosing 5 fighters, I would prefer to be able to select for each fight in the event

I would prefer if I could select fighters for every fight for the whole event

More interactivity between users

I think if we can only choose 5 fighters it will lead to a lot of players making the exact same selection. This would be better if we could choose a winner from each fight.

One user stated that the app could be improved with more interactivity between users, due to time constraints we weren't able to add any more functionality but we recognise that this is a potential weak point of the application and definitely a point we could expand on were we to continue work on it in the future.