Final Year Project Report

Full Unit – Final Report

Advanced Web Development

Chan Kim

A report submitted in part fulfilment of the degree of

BSc (Hons) in Computer Science

Supervisor: Julien Lange

Department of Computer Science

Royal Holloway, University of London

30 March 2023

Declaration

This report has been prepared based on my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count: 14046 words

Student Name: Chan Kim

Date of Submission: 30/MAR/23

Signature: [illegible]

Contents

Abstract

The first ever website was developed by Tim Burners Lee in 1989.[1] With the development of Web browser and web server, protocol called HTTP was defined, and to display information on web browser, HTML was developed.[2] With all these provided as open source, internet has grown with great speed. With the development of CSS, websites could be more diverse in terms of looks, but it was all still static. Dynamic websites started to emerge with the development of JavaScript (JS). Previously, static websites using only HTML had to load everything and download all the sources every time user requested for a new page. Single Page Application has emerged to make things effective, and it could load the static information only once and ask for data needed from server or database when the user ask for it. Today, the web plays a huge part in the software development industry. With the emergence of Progressive Web Applications, demand for advanced web development is expected to grow.[3] Personally, I have a long-term plan to develop a PWA which will replace native apps in the mobile app markets which is predominant at this time.[4] In this project, I aim to develop a website from Frontend to Backend using recent technologies of web development. My main goal is to deliver a web application which provides efficient matchmaking for users looking for football matches to participate in or for users who hold the match looking for players to join. I am a huge fan of football myself, and it is extremely hard for users to find a match to walk in or play regularly unless through friends. This website would help people like me to effectively find a match nearby. If time and energy allow me, the application will also contain social media-like features like "friends" features which will help user to play again with other users and check which upcoming matches other users are playing, and features like "rating users" which will help preventing unfriendly users or users that do not turn up on the match. Furthermore, I aim to implement advanced features of advanced UI and push notification. The push notification feature suits the purpose of the website, as swifter communication is required especially before a few hours/minutes before matches. Academically, my hope is to develop a deeper understanding of modern web applications technology and to understand the pros and cons of different technologies.

Project Specification

Aims: To design and implement a web application for amateur football players finding matches nearby.

Background: Football is usually played by 10~22 people. Amateur players like myself, struggle to find a match to play unless you have an amateur club that plays regularly, or you have enough friends to play together. Even for those who have enough people, it is hard to organise a match against other teams. The aim of the project is to create an online platform that connects football players nearby and hold and manage matches.

A successful project will develop:

(1) Web interfaces for matches and creating matches;

(2) Database storing information of all users and matches;

(3) Account system which allows users to sign up and log in;

(4) Social media-like feature which will connect users to users;

Extensions:

- Push notification feature which will help users to find out new matches and remind the time of the matches in which they are participating.
- Improved usability of the web app
- Rating system for effective use of the platform

Final Deliverables

1. The web app must have a full design, with fully working match participation and match making cycle. It will be implemented in Django (MVC).
2. All codes will be documented.

The app will at least…

3. offer match system with interfaces for match detail and match creation.
4. offer a match search system.
5. offer account system with interfaces for log in and sign in system and profile page.
6. offer database storing all matches and users.

The report will…

7. describe the process of developing the web application
8. exhibit that the requirements are met through uses cases
9. discuss why and why not the technologies/frameworks used are suitable for the web app I am building

Chapter 1: Introduction

Web development is a big part of many software projects, and there are currently many different ways of creating a web application. While some technologies are more trendy and newer than others, it is important to consider and choose the right framework(s) than just going for the fancy modern technologies. Comparing and choosing the right framework can be difficult and even experts have different views on different technologies, so I will focus on learning and reading related literature to choose the best option for the web application I am making. But to do so, it is important to understand the needs of the app I am going to develop.

Through this project, I am developing a web application which connects amateur football players to each other to meet the needs of players who are finding football matches to participate.

In this chapter, I will go through the web app and the main goal of it and discuss which web framework is the most suitable by comparing different state-of-the-art technologies.

1.1 Motivation

I am a huge fan of football myself, and I played football since I was a little kid. Looking back, I played football the most often when I was in high school. It is safe to say that it was because high school encourages students to participate in many different activities, and so it was only my will that was needed to participate in such activities. After graduating high school during my gap year, I struggled to find a group to join or a match to play. By asking friends and family, I could participate in a Facebook group where they play 1 hour session of football on Friday evening. Every Wednesday or so, the group leader post a poll, to check who is coming and who is not. At that time, I thought it is not very efficient as sometimes we had not enough players and the match had to be cancelled few hours before the match. The group did not know how to gather more people from local community. The group leader made an announcement in his church every Sunday to gather more people to join us, but that was not so effective. After I come to Royal Holloway, there were some football matches I could join, but there was not much information about the active sessions or social league. Except when I was in high school, I always struggled to find a football match to play casually or dedicated, even I was fully willing to. I have friends from high school all over the UK, and I could see many of them share the same problem with myself.

Creating an application which helps people like myself will not only help us to participate in more match, but I believe it will also resurge football matches in local communities.

1.2 Aims of the project

The goal of this project is to create a fully functioning web application for amateur football players looking for football matches or other football players to join. The final product at least provides users authenticate system with log-in/sign-up interfaces, profile page where users can view other users information, match page with all the essential information about the match and the ability to join the match and match creation page where the matches will be store in the database when it's created. If all the minimum requirements are fulfilled and the time allows me, additional features will be implemented. This include social media-like feature where users can add each other as 'friends' to follow what match user's 'friend' have and are participating in, rating feature where users can rate other users who participated in the same match for better user experience, push notification feature for users to effectively informed essential information such as match time.

1.3 Comparing Web Development Frameworks

In this section, I am going to compare different web development frameworks and based on the research, the best framework(s) for the project will be chosen.

Web became huge in past few decades, and to meet many diverse needs of the website, many different technologies has emerged. It is almost impossible to investigate every ongoing web framework. First sensible things to do is understand what I am making and how much time I have for developing the app and learning the framework. As I am developing the whole website from scratch, we will investigate both frontend and backend.

1.3.1 Backend Web Framework

In this report, we compare Django, Express.js, Ruby on Rails and CakePHP which are frameworks written in four different languages. All these framework uses the Model View Controller (MVC) architecture. In MVC, an application's data model, user interface, and control logic are separated into three components. The model manages the data of the application and the business rules; the view is responsible for displaying data to the user through an interface; and the controller interprets user inputs and communicates with the model to make the appropriate changes.[5] Figure 1 below shows the structure of the MVC architecture.

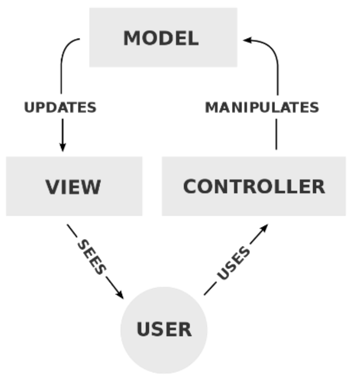


Figure 1. MVC and the interaction between components

All four frameworks are also proven to manage high volumes and are some of most popular choices.

To evaluate these four frameworks, we will need to consider many different factors. In this report, we are particularly interested in usability of templates, content management and popularity.

Terminology can be confusing as different frameworks use different terminologies. In 1), I will use Templates (used in in Django & Express.js) instead of Views (in Rails & CakePHP).

1) Usability of templates (views)

A template contains the static parts of the desired HTML output as well as some special syntax describing how dynamic content will be inserted.[6] The main difference between the frameworks is how they insert the dynamic content into the templates.

In Ruby on Rails, a variable must be declared with '@' to be passed on to the templates (Instance Variable). In the templates, the codes not so intuitive and understandable unless they are used to syntax of Ruby.

In case of Express.js, it has its own templating engine called Pug. Templating engine render input data into output documents through templates. The advantage of using templating engine is that it encourages good code organisation, and the syntax is human friendly. But the downside is that the developer or anyone who works on the templates must learn the new syntax and will also lose some flexibility in templates.

Django also uses its own templating engine, but unlike Express.js, the templates look the same as html documents. The input data are passed from Views and declared with specific syntax in the output documents. But the template language is extremely easy that it can be understood by anyone who knows html but have little knowledge of programming. This is a big advantage for projects involving web designers as they are likely to have little knowledge on programming.

CakePHP is like Ruby on Rails, that it is not so intuitive and understandable unless they are used to syntax of PHP.

2) Content Management

This section focuses on how good the administration interface is used to perform CRUD (create, retrieve, update, delete). This part is important in this project as it can save a lot of time when testing with different data (and managing the content in the future).

Ruby on Rails provides basic admin panel template, but it is too basic and not customisable. It is almost essential that I must use a third-party admin panel or make one myself.

Express.js does not offer default admin panel. Therefore, it must be built from scratch or take advantage of open-source admin frameworks. But this is not the best option as it adds another thing to learn.

Django admin panel is powerful compared to other frameworks. In the Django admin application, developers can use their own models to automatically build a site area that they can use to create, view, update and delete.[7]

In case of CakePHP, it offers basic functionality with some parts that are further customisable.

3) Popularity

Popularity of the framework might seem not so relevant. However, it is important in this case, where I am learning the framework of selection from zero knowledge. Except the case of Django, I also must learn the programming language that the framework is based on. And the popularity of the framework and the programming language is proportional to number of tutorials and helps I could get from online. Statistics & Data, a research analysis and data visualisation project born in October 2019, has created a list of most popular backend frameworks as of January 2022. Django has the highest ranking out of the four, ranked 2nd in the list. Express.js is the next in the ranking at 4th, with Ruby on Rails at 5th and CakePHP going out of the list since 2016. In terms of programming language, according to PYPL index, Python is ranked top of the list, followed by JavaScript at 3rd, PHP at 6th and Ruby at 15th.

In conclusion, Django is overall the best choice for the project considering not much time is given for learning. Also, as I have work experience on html/css, Django's templating engine is most suitable as its templating language is straightforward and the syntax of the templates are identical to the pure html/css documents. In terms of content management, Django's built-in admin panel will be extremely helpful with my project with testing with database and managing the matches and users in the future. As I am already comfortable with Python, I only must learn to use Django framework only.

1.3.2 Frontend Web Framework

In the case of frontend web framework, I am already used to Bootstrap, which is one of the most popular CSS Framework. No advanced features needed in terms of frontend in this project, so it is adequate to use the framework I am most used to.

1.4 Database

A database is an organized collection of structured information, or data, typically stored electronically in a computer system. A database is usually controlled by a database management system (DBMS). Together, the data and the DBMS, along with the applications that are associated with them, are referred to as a database system, often shortened to just database.[8] In this section, I discuss about various types of databases, and justify my choice of the database.

1) NoSQL databases

A NoSQL database – a database that uses not only SQL, is a database that is not primarily built on tables, and generally do not use SQL for data manipulation.[9] It has emerged from challenges in dealing with huge quantities of data with conventional RDBMS solutions could not cope.[10] The problem with NoSQL database is that it typically require more work to query data, as it is less formalised than SQL databases. As we don't require any horizontal scaling or huge amount of data that might affect the speed of the read/write, NoSQL is not chosen this time.

2) Relational databases

Relational database is a database where tables are related to each other. Its schema is defined clearly, and the integrity is guaranteed. It is most widely used at the current stage.

In this project we are going to use SQLite, which is the database used by default in Django.

Chapter 2: Architecture

This part of the report focuses on unpacking the working codes of the project and how they are related to each other. It will mainly be explained in terms of file structure and in terms of MVC architecture.

2.1 File Structure

In this part, I'm explaining what each file does and what they are in Django, in terms of the structure of the files. I will focus on the files I've worked on, which means I will mostly skip the ones that are created by Django by default.

```
futforall/
  └ futforall/
      └ settings.py
      └ urls.py
      └ views.py
      └ consumers.py
      └ routing.py
  └ account/
      └ forms.py
      └ models.py
      └ urls.py
      └ views.py
  └ match/
      └ admin.py
      └ forms.py
      └ models.py
      └ urls.py
      └ views.py
  └ notifc/
      └ urls.py
      └ views.py
      └ models.py
  └ templates/
      └ account/
          └ login.html
          └ profile.html
          └ signup.html
      └ match/
          └ create.html
          └ detail.html
      └ notifc/
```

```
          └ index.html
      └ base.html
      └ base(nav).html
      └ form_errors.html
      └ index.html
```

File structure of the project (the ones I worked on are included only)

There are five main folders in the project – *futforall, account, match, notifc and templates*. *futforall* folder might be a little bit confusing as it has the same name as the project, but by default, Django set the name of the configuration folder as the name of the project.

futforall contains 5 files – *settings.py, urls.py, views.py, consumers.py and routing.py*. *settings.py* is the configuration file for this Django project. This file contains all the configuration of the Django installation. *urls.py* is the file containing the URL declarations. *views.py* is a file containing view. View, or a view function, is a Python function that takes a web request and returns a web response. *views.py* in *futforall* contains view for homepage (i.e., it returns a homepage as a web response). *consumers.py* is a file containing consumers. Consumers are similar to views but for websockets. They are responsible for handling messages sent over the websocket connection. *routing.py* is the file containing routing information for the websocket connection. It maps websocket requests to specific consumers. Together, these files allow for the implementation of a web application that includes both traditional HTTP requests and WebSocket connections.

account is an app in charge of the account/user system of my project. *forms.py* consists of forms for users to sign up and they are made with Django's form classes that are used in the *account* app. I can indeed hardcode the forms in html files, but it is very convenient to use the Django form class. Using forms can be complex. We take numerous different types of data input from a user by preparing a form and displaying it as html, also validate and clean up. Django's form class makes these processes easy and human friendly to read which leads to easy maintenance of the forms. Using Django form helps with preparing and restructuring data to make it ready for rendering, creating html forms for the data, and receiving and processing submitted forms and data from the client.[12] *models.py* consists of models of the app. The Model is the part that manages all tasks related to data in MVC architecture.[13] Intuitively, it is a human-friendly version of database tables. Once you have created a model, you can apply and create tables in the database by migrating. *urls.py* is by default not generated, because in theory, all the URLs can be declared in the configuration *urls.py* file. However, for better organization I have created *urls.py* for each app and included them into the main *urls.py*. *views.py* of account app take care of all the web request related to account (e.g., log-in, sign up etc.).

match is an app that lets users create, view, and participate in the matches. All the files in the *match* app are the same role as account, except one file, *admin.py*. The role of *admin.py* in *match* is to include match models in the admin panel so that we can use it to monitor the matches in our database. *forms.py* consists of forms for users to create matches. *views.py* consists of python functions that take the web request related to matches (e.g., creating a match, checking match details etc.).

notifc is an app responsible for managing notifications for users in the project. The app contains three files – models.py, urls.py, and views.py. The models.py file defines a Notification model with fields for the user, message, whether the notification has been read or not, and a timestamp. The urls.py file defines a single URL pattern that maps to a view function in views.py. When a user accesses the index page, the view function updates all of the user's unread notifications to be marked as read and pass the notifications list as a context to the according template.

template folder is where all the templates are located. A template is a html file containing the static parts of the html output as well as some special syntax describing how dynamic contents will be inserted. Templates are located according to their uses (e.g., login.html located in *template/account*).

2.2 MVC Architecture

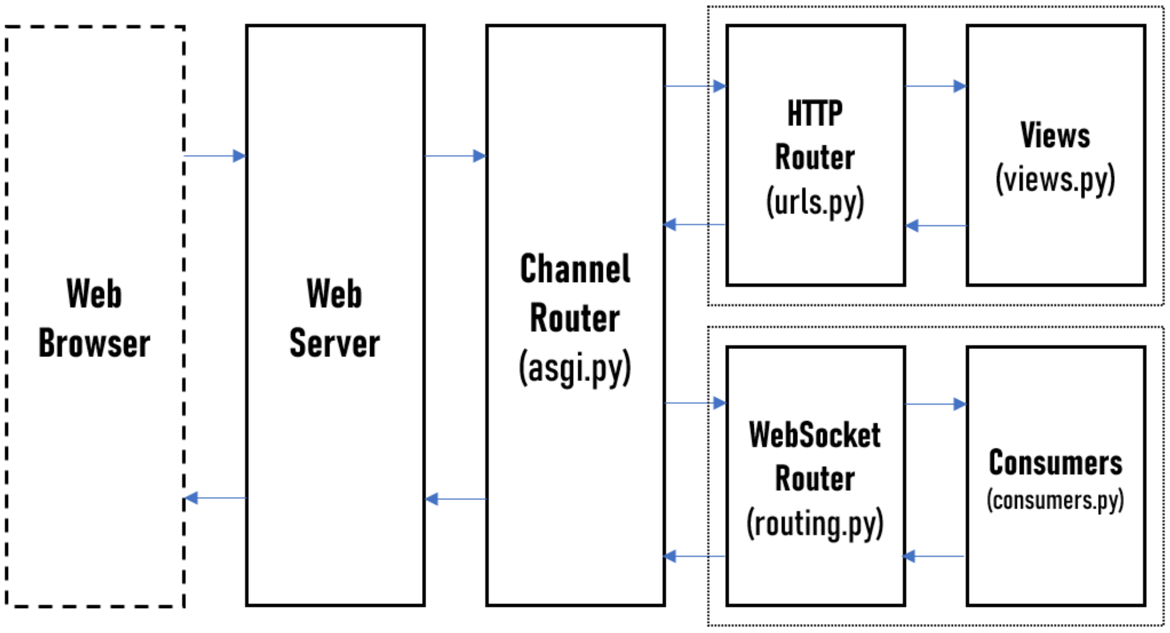


Figure 2. System architecture diagram in terms of HTTP and WebSocket

Figure 2 shows how the system deals with the incoming requests. The channel router is the entry point for incoming requests and direct them to the appropriate channel layers. In case of HTTP requests, the channel router processes them by matching the requested url to a url pattern specified in the system. If there's a matching url pattern, the corresponding view function is called, and it will handle the request and return the response generated to the web browser. In the case of WebSocket requests, the channel router create a separate channel layer from http channel layer. This is because the websocket requests are handled asynchronously. Then the websocket connection will be established. Through the connection, the messages will be sent to the right consumers function by comparing the patterns in the routing.py.

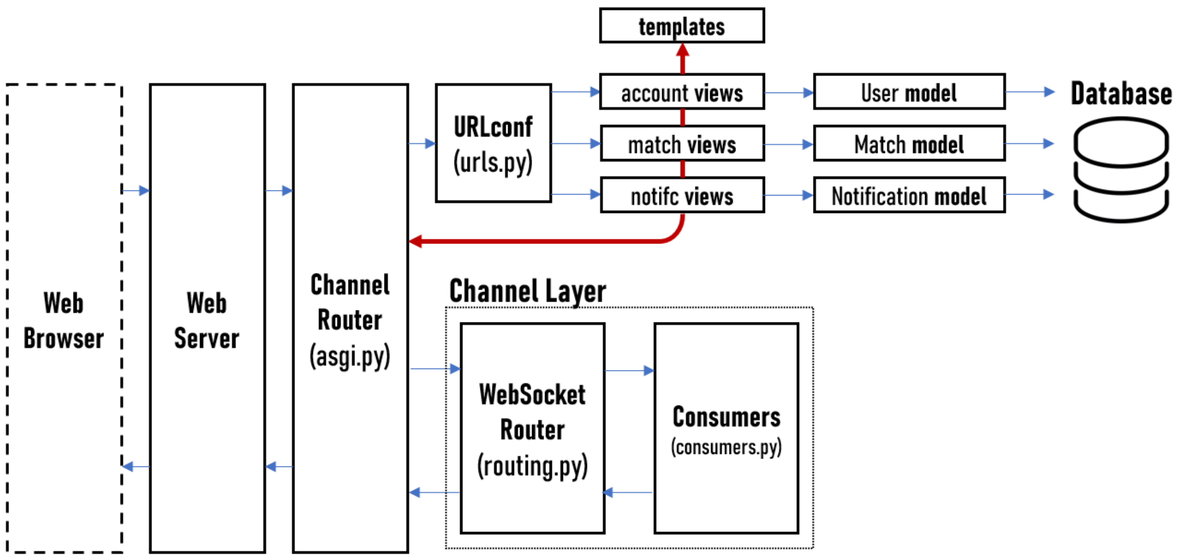


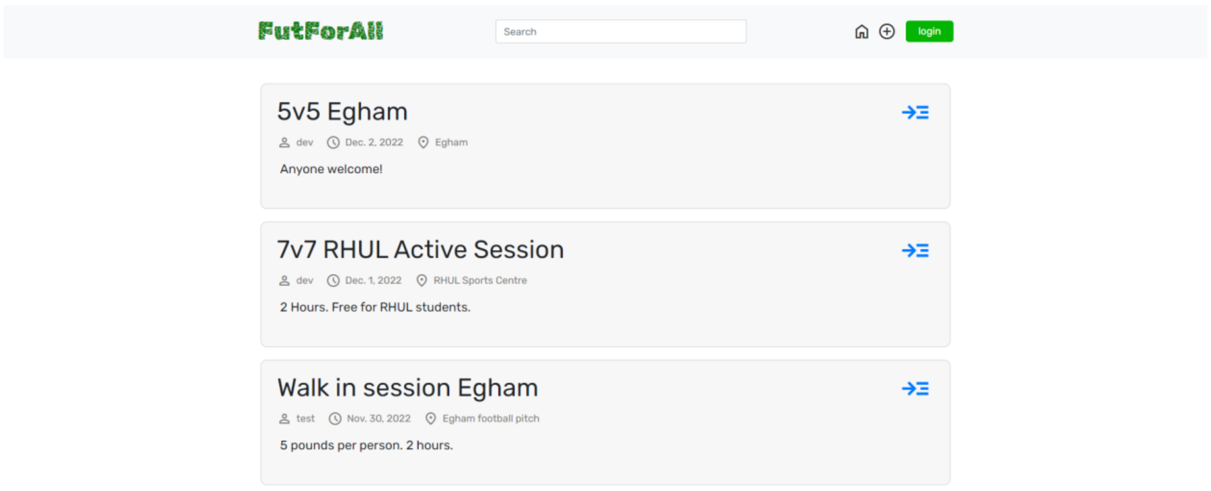
Figure 3. System architecture diagram

Figure 3 shows how the system is structured in terms of the components of Model-View-Controller (MVC) architecture. It can be confusing as the terminologies used in Django are different to the ones used in MVC model generally. The View in MVC is equivalent to Template in Django, and the Controller in MVC is equivalent to the View in Django, and Django shares the same term with MVC for Model.

2.3 Features

In this part, we look into the codes in detail by going through each of the features the web application offers.

When a user enters the web app, the user will see the screen below in Figure 4.



5v5 Egham
dev Dec. 2, 2022 Egham
Anyone welcome!

7v7 RHUL Active Session
dev Dec. 1, 2022 RHUL Sports Centre
2 Hours. Free for RHUL students.

Walk in session Egham
test Nov. 30, 2022 Egham football pitch
5 pounds per person. 2 hours.

Figure 4. Homepage in full screen

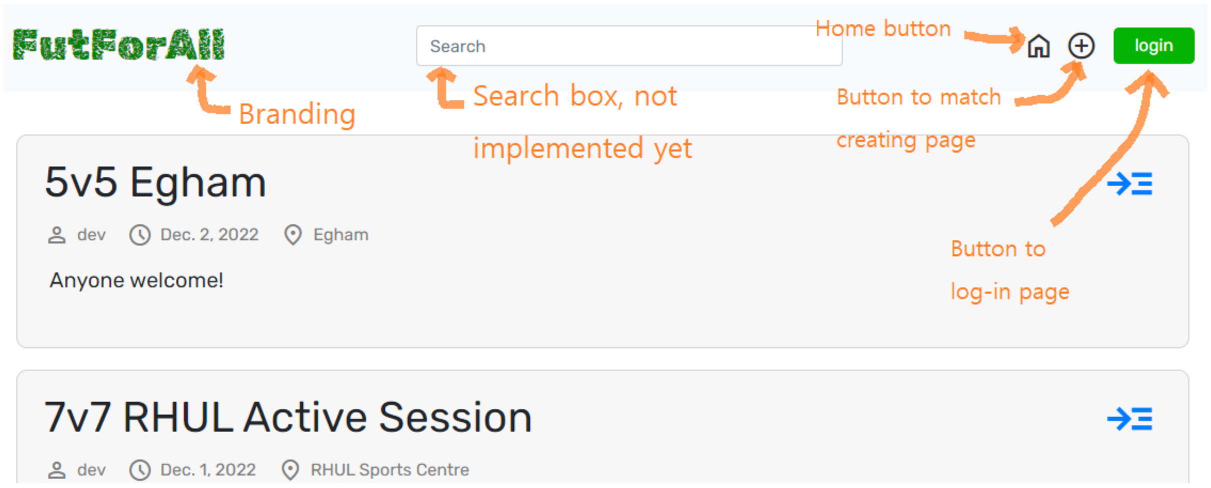


Figure 5. Brief explanation of each component in the homepage

We will divide this section into six parts – homepage, log in page, sign up page, match creating page, match detail page and notification page.

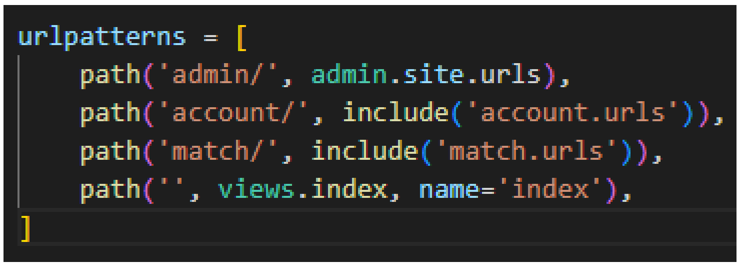
2.2.1 Homepage

When the user connects to the website, the system will try to match the requested url to the url patterns using URLconf. In *futforall/settings.py*, we can see that the root URLconf of the project is set as *urls.py* of the config(*futforall*) folder. (See figure 6)

```
ROOT_URLCONF = 'futforall.urls'
```

Figure 6. root URLconf is set to config urls.py

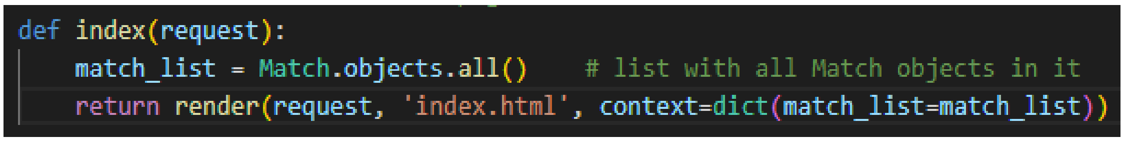
As the user has requested homepage, the requested url pattern is ‘’ (with no character).



```python
urlpatterns = [
    path('admin/', admin.site.urls),
    path('account/', include('account.urls')),
    path('match/', include('match.urls')),
    path('', views.index, name='index'),
]
```

Figure 7. url patterns in futforall/urls.py

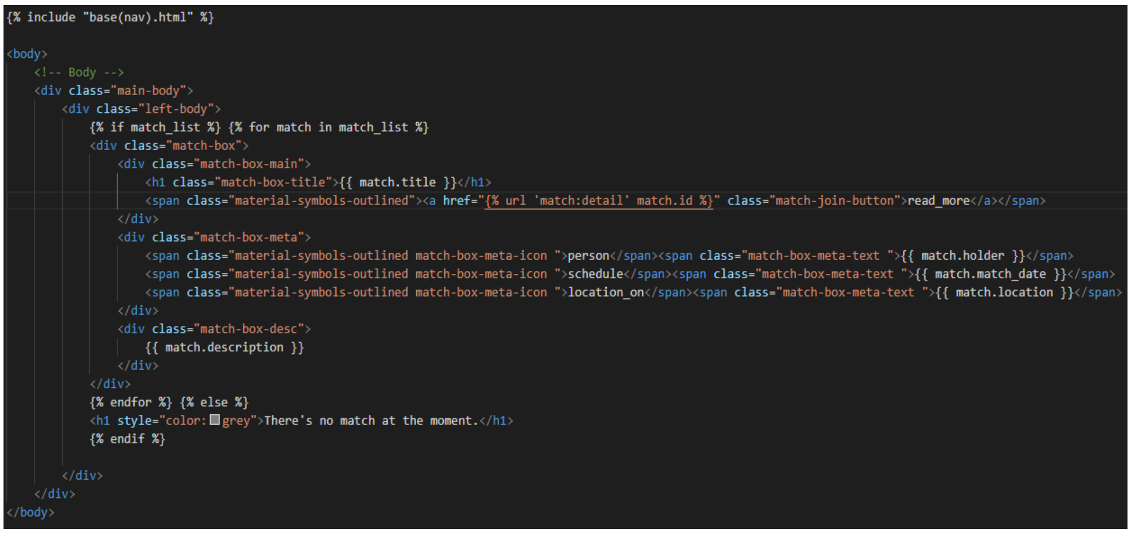
As we can see in figure 5, url pattern with no character is handled by *views.index*.



```python
def index(request):
    match_list = Match.objects.all()    # list with all Match objects in it
    return render(request, 'index.html', context=dict(match_list=match_list))
```

Figure 8. index view

Index view is simply taking request and returning a web response using render function. Render function basically returns the request with which templates to render and the context to be passed on to the template. Context is the dynamic contents to be inserted into the static document. *match_list*, dynamic content being inserted into index.html, is a list of all the matches, which will be used to list all the matches in the homepage, but we can skip this part for now.



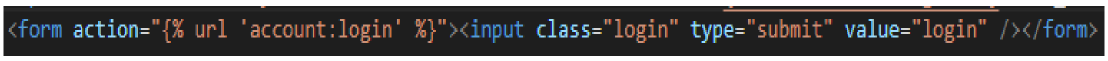
```html
{% include "base(nav).html" %}

<body>
    <!-- Body -->
    <div class="main-body">
        <div class="left-body">
            {% if match_list %} {% for match in match_list %}
            <div class="match-box">
                <div class="match-box-main">
                    <h1 class="match-box-title">{{ match.title }}</h1>
                    <span class="material-symbols-outlined"><a href="{% url 'match:detail' match.id %}" class="match-join-button">read_more</a></span>
                </div>
                <div class="match-box-meta">
                    <span class="material-symbols-outlined match-box-meta-icon ">person</span><span class="match-box-meta-text ">{{ match.holder }}</span>
                    <span class="material-symbols-outlined match-box-meta-icon ">schedule</span><span class="match-box-meta-text ">{{ match.match_date }}</span>
                    <span class="material-symbols-outlined match-box-meta-icon ">location_on</span><span class="match-box-meta-text ">{{ match.location }}</span>
                </div>
                <div class="match-box-desc">
                    {{ match.description }}
                </div>
            </div>
            {% endfor %} {% else %}
            <h1 style="color:grey">There's no match at the moment.</h1>
            {% endif %}

        </div>
    </div>
</body>
```

Figure 9. index.html

Figure 9 is the *index.html* which the index view has asked to render with. Now the user can see the homepage.

2.2.2 Log in page

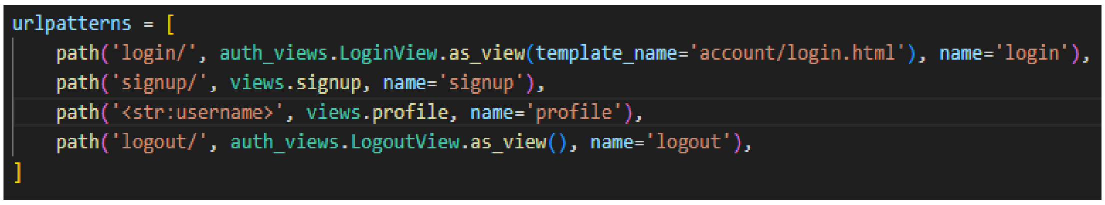
When the user clicks the log in button on the top right corner of the homepage, as we can see in figure 10, it sends a request with url *'account:login'*.



```
<form action="{% url 'account:login' %}"><input class="login" type="submit" value="login" /></form>
```

Figure 10. log in button in base(nav).html

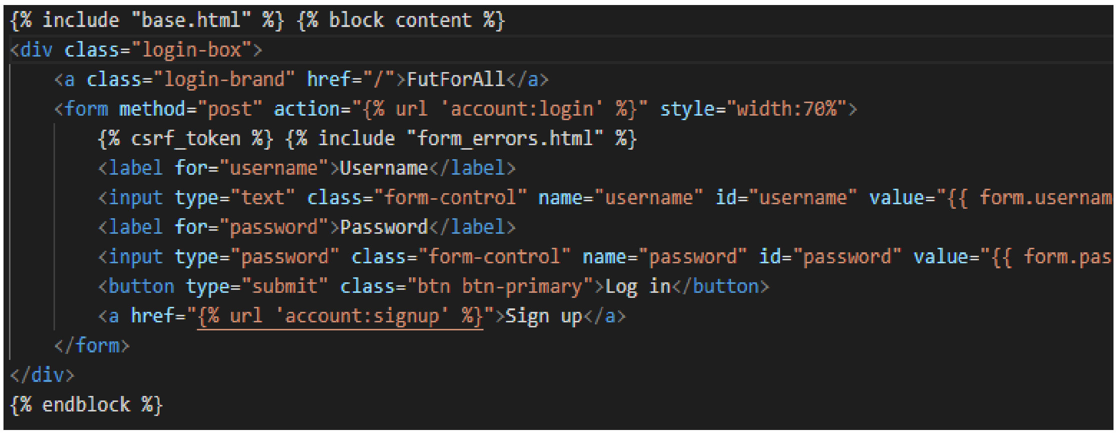
If we go back to figure 7, url containing 'account' is not in the config *urls.py*, but its own *urls.py*.



```
urlpatterns = [
    path('login/', auth_views.LoginView.as_view(template_name='account/login.html'), name='login'),
    path('signup/', views.signup, name='signup'),
    path('<str:username>', views.profile, name='profile'),
    path('logout/', auth_views.LogoutView.as_view(), name='logout'),
]
```

Figure 11. account/urls.py

Now we see the url pattern with name of 'login' which is identical to 'account:login' requested. Unlike how it worked with homepage, the request is not passed on to a view I have created, but a built-in view of Django. The *auth_views* in the code is imported view from Django, which has all the functions to receive the request and return with ready-to-use dynamic contents. Only thing I must do is to tell which templates to render with. Now the user will see the login screen they requested.



```
{% include "base.html" %} {% block content %}
<div class="login-box">
    <a class="login-brand" href="/">FutForAll</a>
    <form method="post" action="{% url 'account:login' %}" style="width:70%">
        {% csrf_token %} {% include "form_errors.html" %}
        <label for="username">Username</label>
        <input type="text" class="form-control" name="username" id="username" value="{{ form.usernam
        <label for="password">Password</label>
        <input type="password" class="form-control" name="password" id="password" value="{{ form.pas
        <button type="submit" class="btn btn-primary">Log in</button>
        <a href="{% url 'account:signup' %}">Sign up</a>
    </form>
</div>
{% endblock %}
```

Figure 12. templates/account/login.html

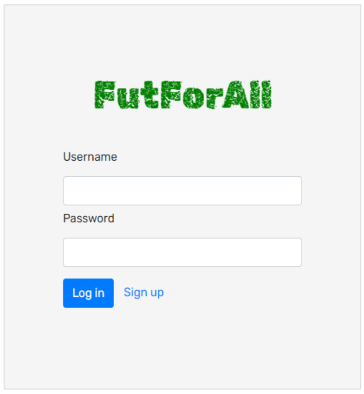
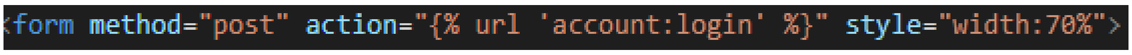


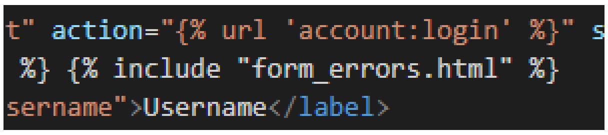
Figure 13. Log in page

When the user input account detail and presses Log in button, the form is submitted with method as POST (request). This request will now be passed on to built-in Django login view and handled by it.



```
<form method="post" action="{% url 'account:login' %}" style="width:70%">
```

Figure 14. templates/account/login.html <form>

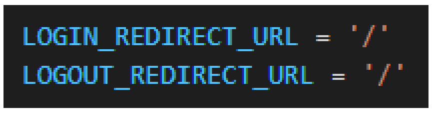


```
t" action="{% url 'account:login' %}" s
 %} {% include "form_errors.html" %}
sername">Username</label>
```

Figure 15. form_errors.html is included in login.html

If we look at figure 15, *form_errors.html* is included in the login form in *login.html*. This is to display and inform the user accordingly when the input detail is faulty or missing. This is the main reason why I chose to use the Django built-in log-in view, as it can respond to all the different kinds of errors in log in form.

The built-in Django login view will handle the request and return adequate context when the inputs are faulty. Otherwise, it will set the current user from 'anonymous' to the authenticated user.



```
LOGIN_REDIRECT_URL = '/'
LOGOUT_REDIRECT_URL = '/'
```

Figure 16. settings.py

After the user is authenticated, the user is then redirected to the homepage as configured in the settings.py. (See figure 16)

2.2.3 Sign up page

Users can connect to the sign-up page by clicking the sign-up button on the log in page. In this section, I will skip how the page is rendered from the user's request as it is the same mechanism as the homepage and the login page.

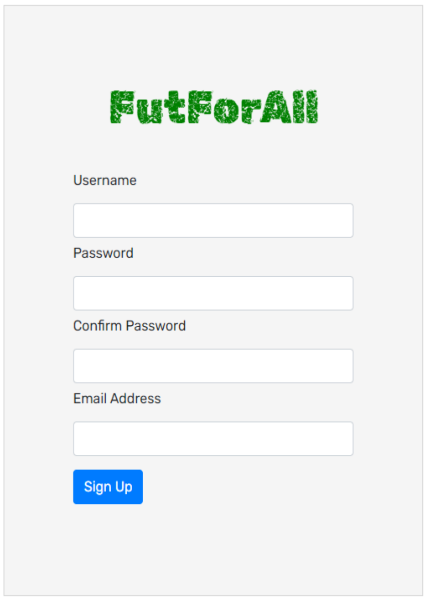
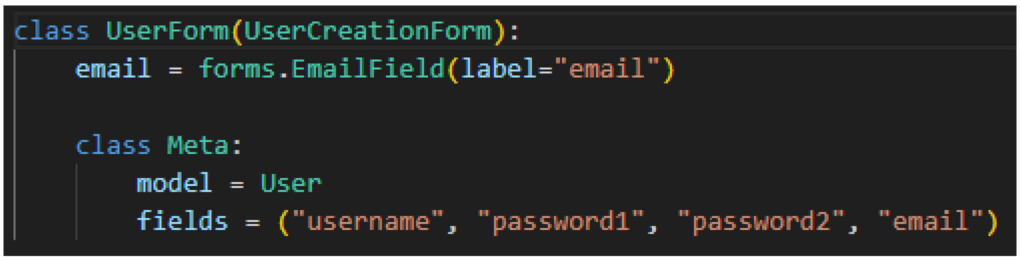


Figure 17. Sign up page

Similar to the log in feature where I chose to use the Django built-in login view, I chose to use the Django built-in sign-up view, but it's a little bit different for sign up feature. I could use the Django built-in login view without having the view function declared by myself. However, in case of sign-up feature I chose to create my own Django form class to create sign up form. This is because I potentially have to add additional attributes which are not included in the built-in sign-up view.

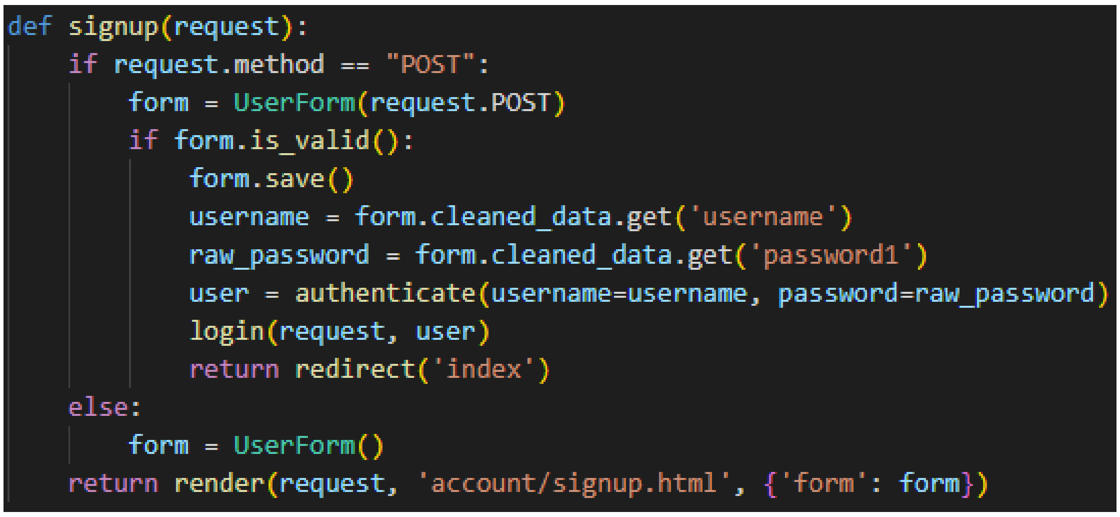


```python
class UserForm(UserCreationForm):
    email = forms.EmailField(label="email")

    class Meta:
        model = User
        fields = ("username", "password1", "password2", "email")
```

Figure 18. sign up form in account/forms.py

For now, I have pretended that email address is the attribute I want to add even if the email address is already included in the built-in sign up view. The sign-up form is created by inheriting the built-in sign-up form which is *UserCreationForm*.

As I have created my own sign-up form, I have to create my own view for the sign-up feature.



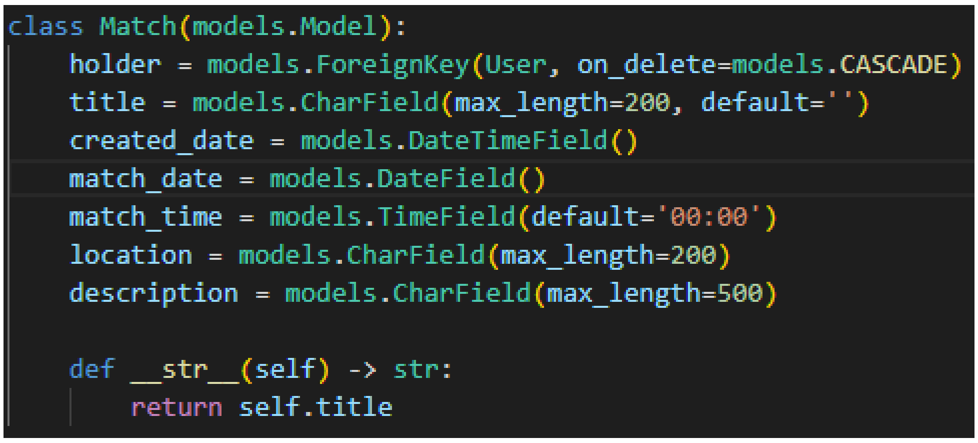
```python
def signup(request):
    if request.method == "POST":
        form = UserForm(request.POST)
        if form.is_valid():
            form.save()
            username = form.cleaned_data.get('username')
            raw_password = form.cleaned_data.get('password1')
            user = authenticate(username=username, password=raw_password)
            login(request, user)
            return redirect('index')
    else:
        form = UserForm()
    return render(request, 'account/signup.html', {'form': form})
```

Figure 19. sign up view in account/views.py

Sign up view take the request and check if the user has requested to sign up. If that is not the case, the view returns the render method with signup template and passes the signup form. In the case of the user requested to sign up, signup view receives the form completed by the user and checks if the form is valid. *Is_valid()* is a built-in method in the Django built-in signup view. If the completed form is not valid, the signup view will still render the signup.html, but *form_errors.html* will inform user what the problem is. If the form is valid, then first the form is saved, and then the input data is cleaned. Then we authenticate the user and redirect the user to the homepage.

2.2.4 Match detail page

Unlike user authenticating system, match is something I had to create from scratch. First thing I did was to create a model of match.



```python
class Match(models.Model):
    holder = models.ForeignKey(User, on_delete=models.CASCADE)
    title = models.CharField(max_length=200, default='')
    created_date = models.DateTimeField()
    match_date = models.DateField()
    match_time = models.TimeField(default='00:00')
    location = models.CharField(max_length=200)
    description = models.CharField(max_length=500)

    def __str__(self) -> str:
        return self.title
```

Figure 20. Match model in match/models.py

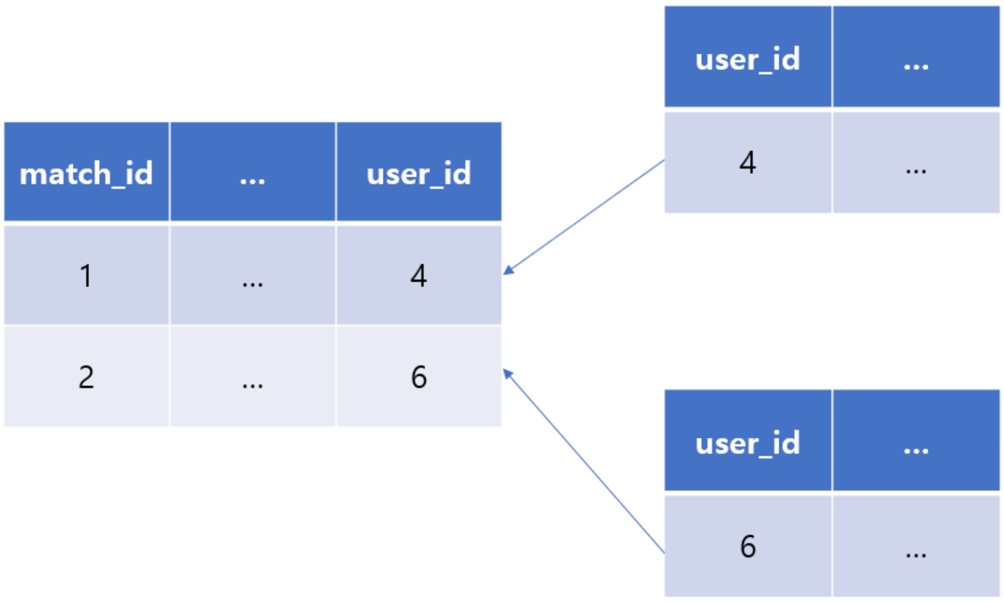
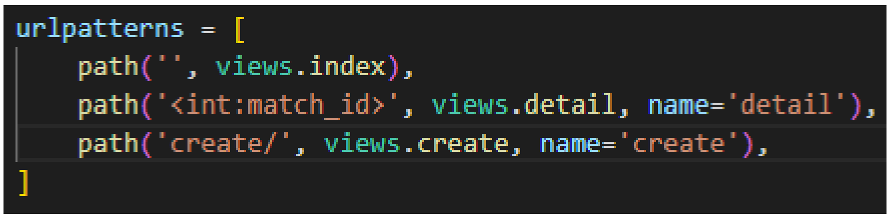


Figure 21. Relation between user and match

A match consists of a holder – the user who has created the match, a title, created date, match date, match time, location of the match, and a description. Holder is a foreign key as a match is created by a user (figure 21). I have also explicitly set the match to be deleted when the user created the match is deleted. On a side note, there will be more attributes added in the future (e.g., participants, duration etc.). *__str__* method has been created to help myself distinguish between matches when I try out the feature or debug with python shell.

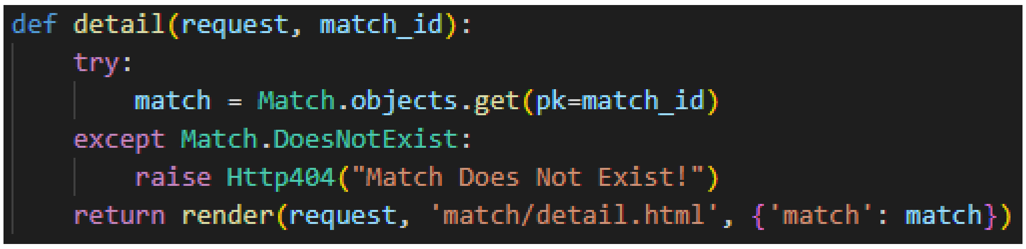
Now we look at the *urls.py* of the *match* app.



```
urlpatterns = [
    path('', views.index),
    path('<int:match_id>', views.detail, name='detail'),
    path('create/', views.create, name='create'),
]
```

Figure 22. url patterns in match/urls.py

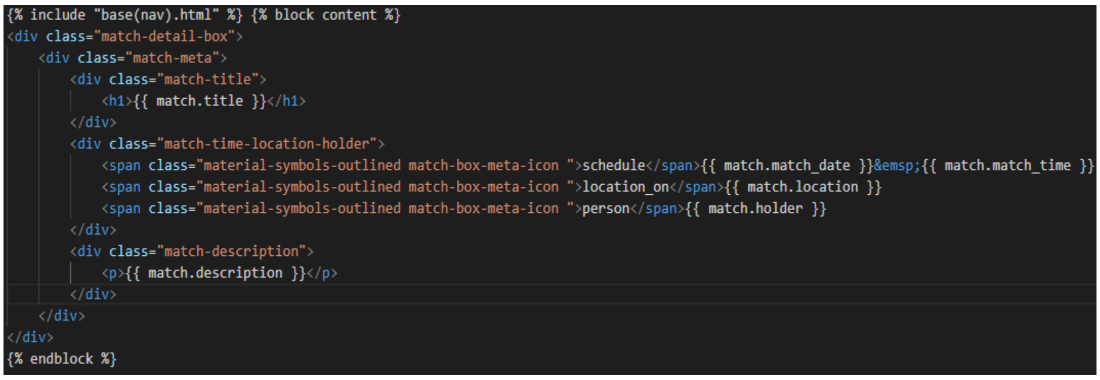
We can see the second url pattern is expecting integer variable. This is to return a match page of match id in the url. The *match_id* is passed to *views.detail* via url.



```python
def detail(request, match_id):
    try:
        match = Match.objects.get(pk=match_id)
    except Match.DoesNotExist:
        raise Http404("Match Does Not Exist!")
    return render(request, 'match/detail.html', {'match': match})
```

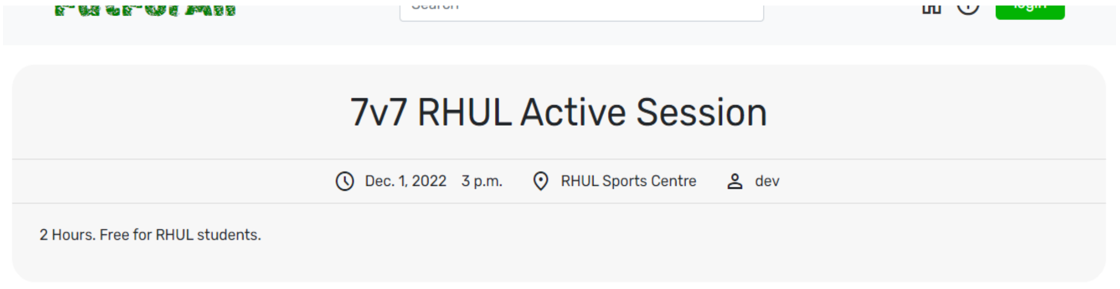
Figure 23. detail function in match/views.py

The detail view receives request and the *match_id* as arguments. First the view tries to find a match with the *match_id* received. If the match exists in the database, then it is assigned to a variable, otherwise throw a 404 error. It returns the render method with *detail.html* template and pass the match variable on to it.



```html
{% include "base(nav).html" %} {% block content %}
<div class="match-detail-box">
    <div class="match-meta">
        <div class="match-title">
            <h1>{{ match.title }}</h1>
        </div>
        <div class="match-time-location-holder">
            <span class="material-symbols-outlined match-box-meta-icon ">schedule</span>{{ match.match_date }}&emsp;{{ match.match_time }}
            <span class="material-symbols-outlined match-box-meta-icon ">location_on</span>{{ match.location }}
            <span class="material-symbols-outlined match-box-meta-icon ">person</span>{{ match.holder }}
        </div>
        <div class="match-description">
            <p>{{ match.description }}</p>
        </div>
    </div>
</div>
{% endblock %}
```

Figure 24. detail.html simply displays all the information of the match



7v7 RHUL Active Session

Dec. 1, 2022 3 p.m. RHUL Sports Centre dev

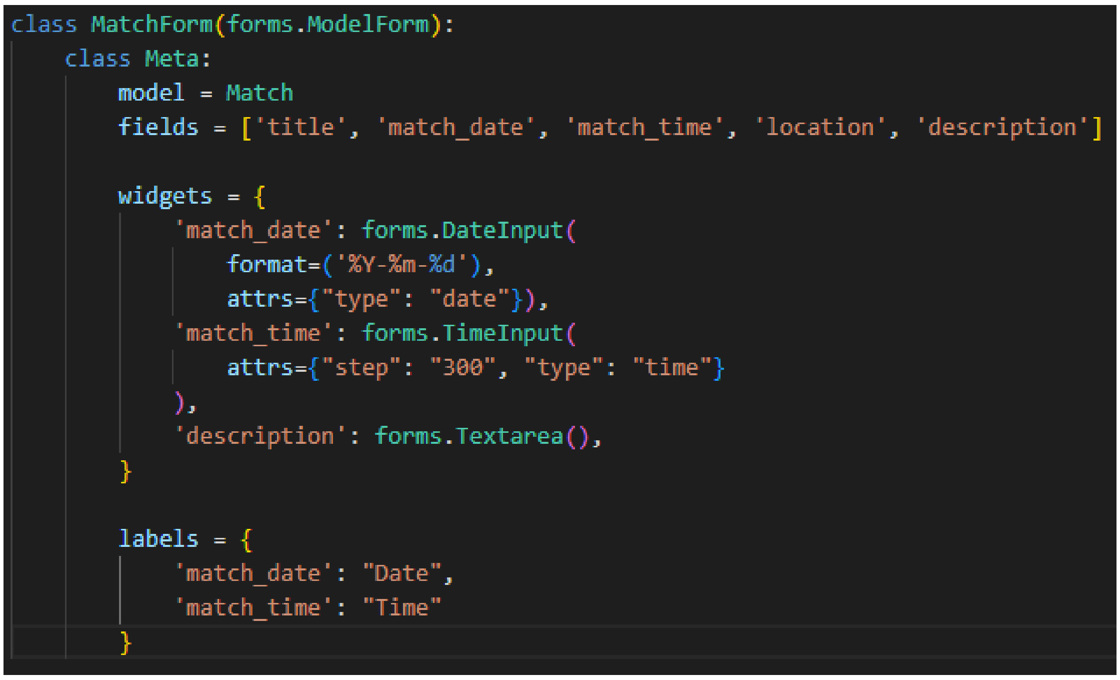
2 Hours. Free for RHUL students.

Figure 25. ../match/5 (match detail page)

2.2.5 Match creating page

By pressing the + button on the top right corner in homepage, user can connect to match creating page (../match/create/).

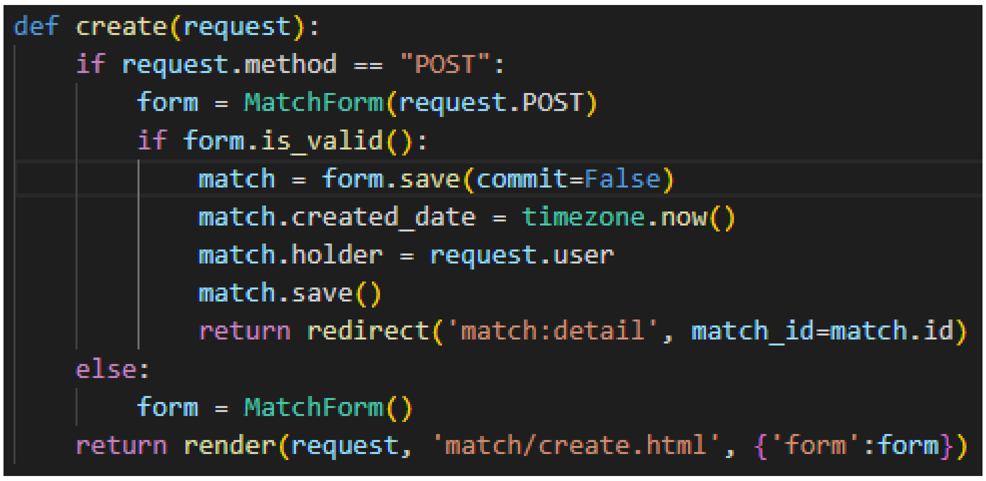
For match creation form, I've used a Django built-in *ModelForm*. *ModelForm* is useful for form of existing model. As I have *match* model already, this was the sensible choice.



```python
class MatchForm(forms.ModelForm):
    class Meta:
        model = Match
        fields = ['title', 'match_date', 'match_time', 'location', 'description']

        widgets = {
            'match_date': forms.DateInput(
                format=('%Y-%m-%d'),
                attrs={"type": "date"}),
            'match_time': forms.TimeInput(
                attrs={"step": "300", "type": "time"}
            ),
            'description': forms.Textarea(),
        }

        labels = {
            'match_date': "Date",
            'match_time': "Time"
        }
```

Figure 26. MatchForm in match/forms.py

With *ModelForm*, first we let it know which model this form is for. And we specify which input fields are to be included. Then I have specified what formats I want for *match_date* and *match_time*. Also, for users to intuitively realise that description allows longer input, we use textarea which is a bigger input space for texts. Then I changed the label of *match_date* and *match_time* to more human-friendly terms.



```python
def create(request):
    if request.method == "POST":
        form = MatchForm(request.POST)
        if form.is_valid():
            match = form.save(commit=False)
            match.created_date = timezone.now()
            match.holder = request.user
            match.save()
            return redirect('match:detail', match_id=match.id)
    else:
        form = MatchForm()
    return render(request, 'match/create.html', {'form':form})
```

Figure 27. create view in match/views.py

Create view is similar to the sign-up form, but there's a difference when actually creating the match in database. If the input value for match is valid, first the input values are assigned to a variable *match*. But I specified not to actually save it in the database just yet. This is because there are other attributes that need to be set before inserting into the database. Without this, the user will encounter an error that some essential attributes are not set, and we don't want to make created date and the holder to optional (holder is foreign key so this is not an option anyway). Match created date is right now when the match is created, and the holder is the user who has created this match. After assigning all the attributes, we finally save the match to the database and redirect the user to the match detail page of the match just created.

2.2.6 Notification page

By clicking the bell button on the navigation bar (Figure 28), the user can connect to the notification page.

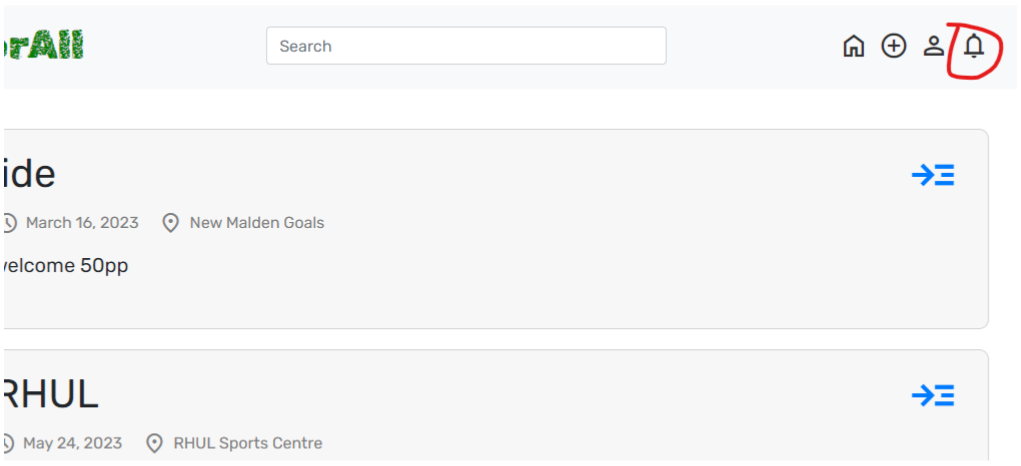
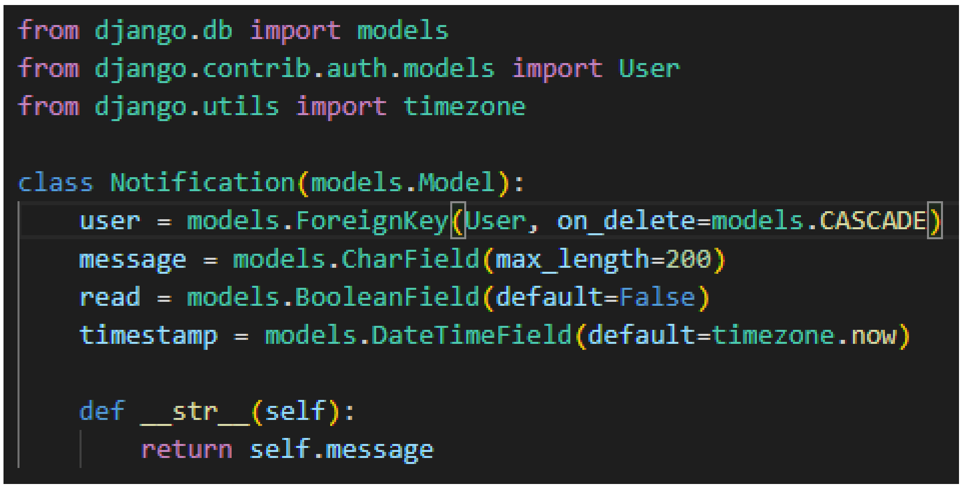


Figure 28. notification button

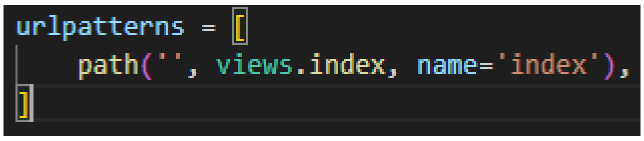


```
from django.db import models
from django.contrib.auth.models import User
from django.utils import timezone

class Notification(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    message = models.CharField(max_length=200)
    read = models.BooleanField(default=False)
    timestamp = models.DateTimeField(default=timezone.now)

    def __str__(self):
        return self.message
```

Figure 29. notifc/models.py

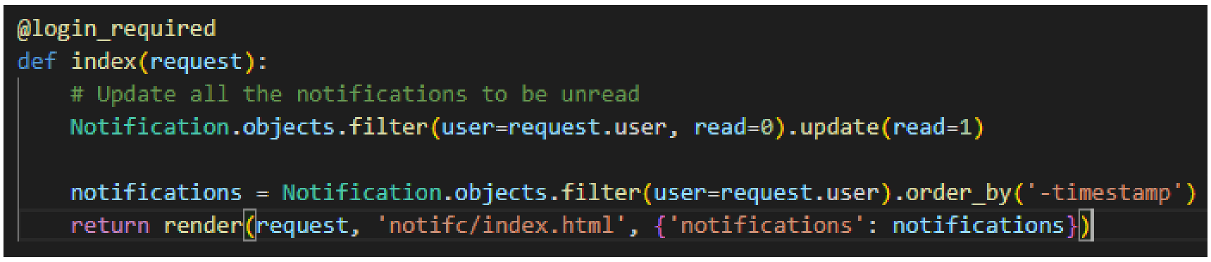
The model for notification has four fields – *user, message, read* and *timestamp*. *User* is a foreign key to the built-in user model. This means every notification is bound to only one user. *Message* is a string field that stores the text of the notification message. *Read* is a Boolean field which is true when user has read the notification and false otherwise. *Timestamp* is a datetime field that records the time the notification was created. *__str__* method simply returns the notification message when the model instance is printed as a string.

Next, we look at the *urls.py* for *notifc*.



```
urlpatterns = [
    path('', views.index, name='index'),
]
```

Figure 30. notifc/urls.py



```
@login_required
def index(request):
    # Update all the notifications to be unread
    Notification.objects.filter(user=request.user, read=0).update(read=1)

    notifications = Notification.objects.filter(user=request.user).order_by('-timestamp')
    return render(request, 'notifc/index.html', {'notifications': notifications})
```

Figure 31. notifc/views.py

It simply has one url pattern, which maps the 'notifc/' to the index view of notifc app. The index view function is decorated with *@login_required* which is pretty much self-explanatory. Only the authenticated users can access this view. When a user connects to the notification page, this view is called, and the function first updates all the notifications that are unread and set the *read* attribute to 1 (True). Then the function queries the db to get all the notifications and order it by -timestamp which simply means order from the most recent ones. And at last, it renders *notifc/index.html* with notifications context variable which contains the notifications fetched from the db and sort in order of recent to old ones.

Chapter 3: Software Development Methodology

This chapter explains what software development methodology I've chosen and the reason behind it. And I cover the process of the development in terms of the development methodology and how it is applied in the process. Limitations of the methodology in the process and the advantages and disadvantages of the methodology by the experiences throughout the process of developing will also be covered.

3.1 Agile

While there's no agreement on what the concept of "agile" actually refers to [14], there are some clear characteristics that associate with it. The background of the emergence of this approach is indeed the demand of the business community and the rapidly growing and volatile online software industry.[15] By understanding this, we know the focal value of the Agile software development methodology. The demand from the market could be summarized into "speed" and "demand". As the name "agile" implies, swift delivery of the product is one of the key concepts. The methodology does this by setting smaller goals and delivering from the minimum requirements. By the distribution from the smaller components – which would be considered early stage of the development in more traditional approaches, it enables the project to react and act accordingly to the demands and feedbacks of the customer. The agile movement also emphasizes the relationship and community of the developers as opposed to the traditional institutionalized processes and tools.[16] Below are the focal values of the methodology summed up to four bullet points.

- **Individuals and interactions** over processes and tools
- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan [17]

The reason I chose this methodology is mainly because the process of agile is similar to how I work as it is heavily function/working software oriented. Moreover, it is encouraging and motivating to test and use the deploy-ready software throughout the process of the development. The relationship and communal characteristics of the methodology does not apply in this project as I'm the only developer working on the project, thus I focus on the planning and developing side of the methodology.

3.2 Process and review

3.2.1 Disadvantages

The first task I had to do before diving into technical part of the project was to plan the process. It was obvious that in big chunks, I have to divide the product into two parts – 1) match, account app with minimum functionalities and 2) notification system and social network like features – as the most basic goal of the web app is to help users find matches to participate in. However, it was more complicated when it comes to planning it into smaller parts. In the process, many smaller steps were changed in order and implementation in more detail. An example of this can be found from my initial plan of the project. In the project plan I've submitted at the beginning of the year, in week 4 of section 2.1 (timeline for term 1), my plan was to complete the homepage and creates all the links to the pages in the menu (which is referring to the navigation bar). This was a bad approach looking back now, as it is not the most natural way of development process that would help the process the development. It was planned in the view of a user of a website with very little experience in web development. It is more sensible to divide parts strictly in terms of functionalities – which in Django is often referred to as an app – and merge them accordingly afterwards. In the planning phase of the project, I was not in the understanding of this, and thought about how to merge the functionalities first then the actual functionalities. This could feel like a trivial difference, but it actually does make a big difference as when planning how to connect different apps, I have very little knowledge of how the apps would turn out eventually. This could cause bigger problems in bigger software and when working with bigger teams where communication actually matters, but I still had some minor problems regarding this issue. An example of this mistake is that I spent significant amount of time planning the homepage and how should the apps be connected with each other. When I actually finished most of the apps, the connection I initially planned were almost changed in every aspect which means I could save significant amount of time by focusing on each functionality as a Django app.

It is also worth to note that the continuous changes and updates to the software can be a downside in this specific situation. Most changes and updates come from the feedbacks and use cases, and in this case, I am the only tester/user, and the feedbacks are solely from myself. This can lead to the application tilting to suit people similar to me or also my environment. This is a noteworthy point only because I am the only developer and also an only tester, which is not the most ideal way to develop and test a software.

As mentioned in the four focal values of the Agile, the methodology places a higher value on working software over comprehensive documentation. This has certainly been the case for this project as well, as the documentation for the project has been on the back burner. While documentation is important when it comes to maintaining the software, it has been a less significant problem in this case as I am the only developer.

3.2.2 Advantages

There was also a clear advantage that helped the project. After completing the match and account part of the web application, I tested the software multiple times – which would have been the role of the user of software in real life – which led to many major and minor updates of the first version of the match and account app. By having feedbacks and updates before going into the next phase of the development, the next feature to be implemented could be worked on top of that. This is significant as the feedbacks from the user could lead to major changes to the software, which might also require change to the new features implemented if it's applied before testing and feedbacks. Also on a side note, as the methodology involves regular feedbacks and testing, it reduces the risk of issues and failure vastly. This is an important note, since it naturally motivates me from smaller apps to larger ones, and the failure on the process could lead to avolition. As I'm the only developer, if I struggle, the whole process can be delayed.

3.2.3 Conclusion

Below I've concluded the advantages and disadvantages of the methodology by my experience throughout the process of developing *Futforall.*

Advantages

- Ease of planning the whole project
- Testing the software multiple times led to major and minor updates that helped in the development process
- Regular feedback and testing reduce the risk of issues and failure significantly
- Motivates the developer to work on smaller apps and reduces the risk of avolition

Disadvantages

- Changes and updates may lead to the application tilting towards the developer's preferences and environment when the developer is the only tester
- Limited documentations

Chapter 4: Development and Deployment

This part of the report explains the technical part of the process of the development. This includes from setting up the development environment to programming the web application and solving the problems I faced during the procedure. First, I briefly talk about how to set up the environment, and about the tools I chose. And then I talk about the process of the creating the web application, focusing more on the problems I faced and how I tackled it rather than listing what I did. On a side note, everything is in terms of Windows.

4.1 Setting up development environment

- **virtual environment**

When working on a project, it is desirable to create a separate virtual environment for the project. As different projects need different libraries, and sometimes even different versions of the same library, virtual environments help to organise the libraries in different projects.

Setting up a virtual environment in Python is easy. Python has a module called venv, which supports creating lightweight virtual environments. It can be done by simply running the code below in the command line.

```
python3 -m venv /path
```

The path above will be the path of the new virtual environment. Running the code above will create necessary files for the environment to work.

To activate the virtual environment, simply run the file named `activate.bat` in the folder `Script` in the virtual environment folder. Below is the code I run.

```
.\Script\activate
```

Once the virtual environment is activated, you will see the name of the virtual environment in bracket at the left end of the command line. Below is my virtual environment for the project activated.

```
(futforall) C:\Users\...\futforall>
```

- **Django**

Installing *Django* is easy using *pip*, a built-in python package manager. Running the below code will install Django.

```
py -m pip install Django
```

Create a Django project:

```
django-admin startproject <project-name>
```

Create a Django app:

```
django-admin startapp <app-name>
```

Django app is a web application with some functionality – e.g. a blog system, small poll app. A project is a collection of configurations and apps for a particular website. A project usually contains multiple apps. An app can be in multiple projects.[18]

All the apps created should be included in the *INSTALLED_APPS* in the *settings.py* file of the project for the app to function properly. Below is an example.

futforall/settings.py:

```
INSTALLED_APPS = [
       'some_app',
       '…',
]
```

Running the Django development server:

```
python manage.py runserver
```

- **Git**

Git is a version control system that keep the project organised and riskless. I used Git as the college use Gitlab.

- **Bootstrap**

Bootstrap is a well-known CSS framework that is free and open-source. Using *bootstrap* is very simple.

```
<link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/bootstrap@4.3.1/dist/css/bootstrap.min.css"
integrity="sha384-
```

```
ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T"
crossorigin="anonymous">
```

Code above should be included in the *<head>* part of the template where the bootstrap classes are called.

- **Fonts**

In the website I used free Google font called *Rubik*. It is very easy to import the font to be used in the website. I simply added the below code to the *<head>* in the template where styles are declared.

```
<link rel="stylesheet" href="https://fonts.googleapis.com/css?family=Rubik">
```

- **Django Channels**

Django Channels is a library for *Django* that allows the Django web application to support websockets and other asynchronous protocols. Django Channels uses Asynchronous Server Gateway Interface *(ASGI)* protocol to handle asynchronous communications. This library is used for real-time notification feature in this project. Installing *Channels* is done by running the code below.

```
pip install channels
```

Once it's installed, it should be added to *INSTALLED_APPS* in settings.py file of the project and define the async consumers and routing rules.

futforall/settings.py:

```
INSTALLED_APPS = [
    'channels',
    '…',
]

ASGI_APPLICATION = "futforall.asgi.application"
```

The *ASGI_APPLICATION* here is pointing to the application in asgi.py of project config folder.

futforall/asgi.py:

```
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'futforall.settings')
...
application = ProtocolTypeRouter({
    "http": get_asgi_application(),
    "websocket": AuthMiddlewareStack(
        URLRouter(
            futforall.routing.websocket_urlpatterns
        )
    ),
})
```

This file sets up an ASGI application. We tell the system the location of the project settings file, and in *application*, we simply tell the system where to look for the web socket patterns.

4.2 Development process

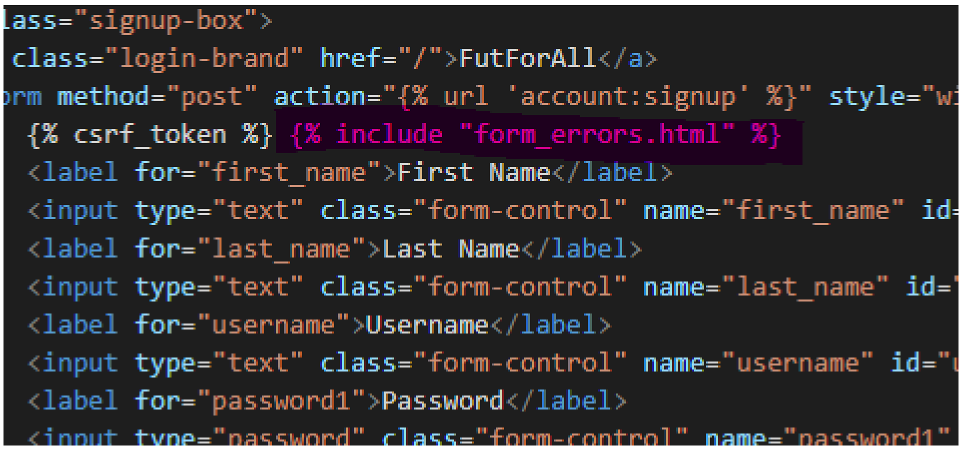
4.2.1 *account*

Account app is in charge of the authentication feature of the software and the profile feature that is linked to other apps in the software. Below are the files that are directly related to the account app.

```
└ account/
    └ forms.py
    └ models.py
    └ urls.py
    └ views.py
└ templates/
    └ account/
        └ login.html
        └ profile.html
        └ signup.html
```

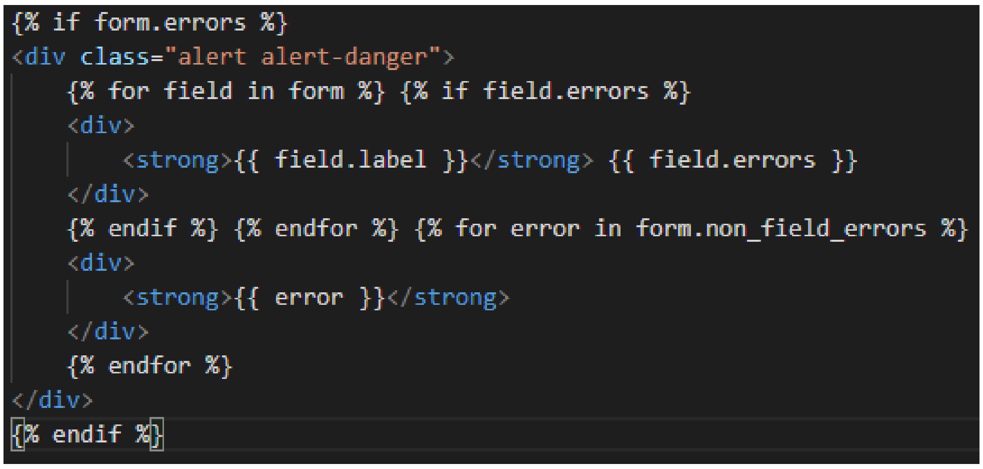
On a side note, the first step of creating this web application was setting up an authentication system. This is a sensible thing to do as out of all the functionalities I have to implement for the project, the authentication system is the only app that is independent of other apps.

For authentication system, I used Django's built-in system as it contains all the functions I need and it is efficient. The authentication system itself might be simple, but there are some essential features like passwords requirements in the authentication system that is crucial but can be troublesome to begin from scratch and there are many minor things that has to be implemented. Below is an example.



```
lass="signup-box">
 class="login-brand" href="/">FutForAll</a>
orm method="post" action="{% url 'account:signup' %}" style="wi
  {% csrf_token %} {% include "form_errors.html" %}
  <label for="first_name">First Name</label>
  <input type="text" class="form-control" name="first_name" id=
  <label for="last_name">Last Name</label>
  <input type="text" class="form-control" name="last_name" id="
  <label for="username">Username</label>
  <input type="text" class="form-control" name="username" id="
  <label for="password1">Password</label>
  <input type="password" class="form-control" name="password1"
```

Figure 32. signup.html

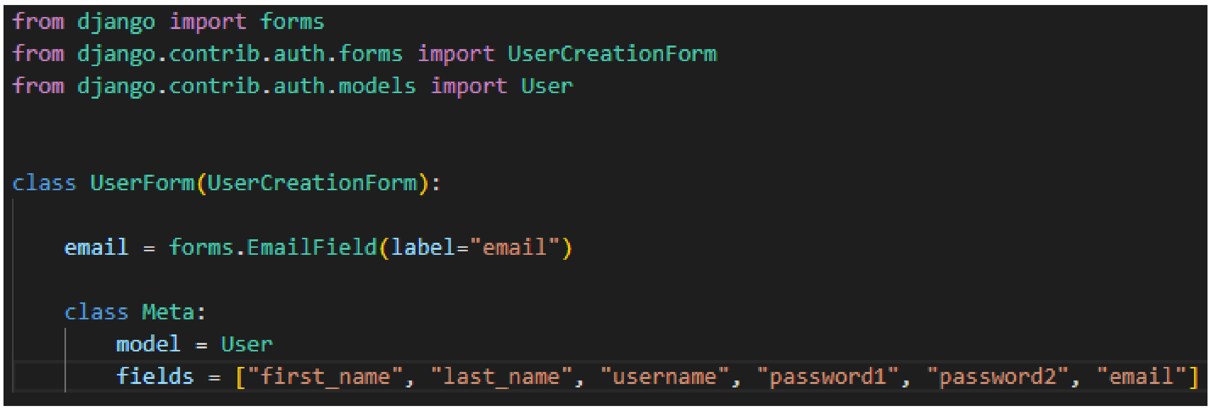


```
{% if form.errors %}
<div class="alert alert-danger">
    {% for field in form %} {% if field.errors %}
    <div>
        <strong>{{ field.label }}</strong> {{ field.errors }}
    </div>
    {% endif %} {% endfor %} {% for error in form.non_field_errors %}
    <div>
        <strong>{{ error }}</strong>
    </div>
    {% endfor %}
</div>
{% endif %}
```

Figure 33. form_errors.html

I simply include the *form_errors.html* in the *signup.html*, and in *form_errors.html* I simply include the code that calls the error that the user has entered incorrectly. All the logic behind it is done by *Django* library, and I just have to call it. If it was the case that I require more specific sign-up information or some specific feature, it might not be a good idea to use the built-in authentication system by Django, but in this case, I only need the simplest features for user authenticate system.

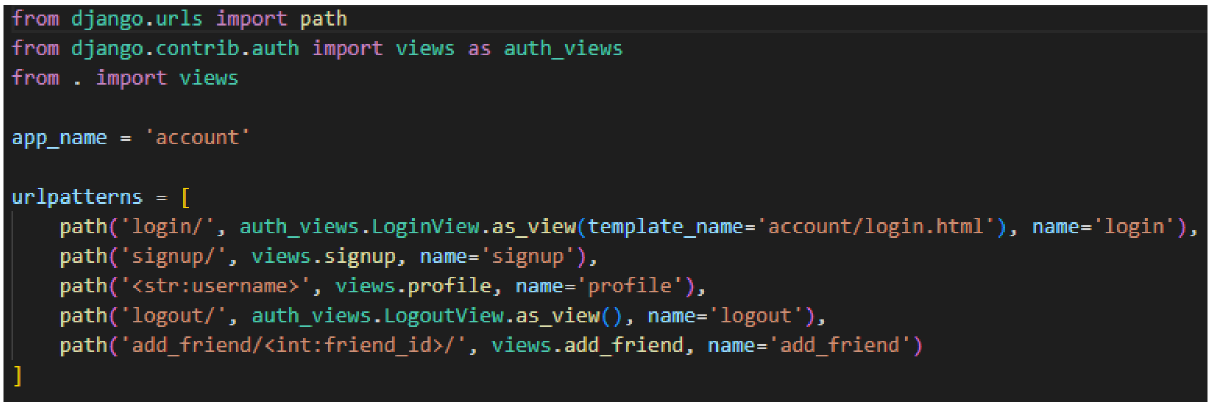
For the sign-up system, I used Django forms rather than just using the html form on its own. This is a good idea as the Django form can automatically process the data such as converting user inputs into the correct data types or saving forms to the db.



```python
from django import forms
from django.contrib.auth.forms import UserCreationForm
from django.contrib.auth.models import User


class UserForm(UserCreationForm):

    email = forms.EmailField(label="email")

    class Meta:
        model = User
        fields = ["first_name", "last_name", "username", "password1", "password2", "email"]
```

Figure 34. account/forms.py

Also on a side note, the *UserCreationForm* surprisingly only had three fields: username, password1 and password2 (for validation). Therefore, I had to manually include the email, first name and last name. (Figure 34)



```python
from django.urls import path
from django.contrib.auth import views as auth_views
from . import views

app_name = 'account'

urlpatterns = [
    path('login/', auth_views.LoginView.as_view(template_name='account/login.html'), name='login'),
    path('signup/', views.signup, name='signup'),
    path('<str:username>', views.profile, name='profile'),
    path('logout/', auth_views.LogoutView.as_view(), name='logout'),
    path('add_friend/<int:friend_id>/', views.add_friend, name='add_friend')
]
```

Figure 35. account/urls.py

One thing to mention about the URLs of account app is, the url for user profile page. Instead of using the *user_id* in the url, I decided to use the username. This way it is more intuitive for users but also it helps with SEO since search engines often prioritise URLs with relevant keywords, which is why most social media platforms have usernames in the url instead of id. However, also have to note that there are more things to consider such as the username cannot contain specific characters that can't be used in URLs.

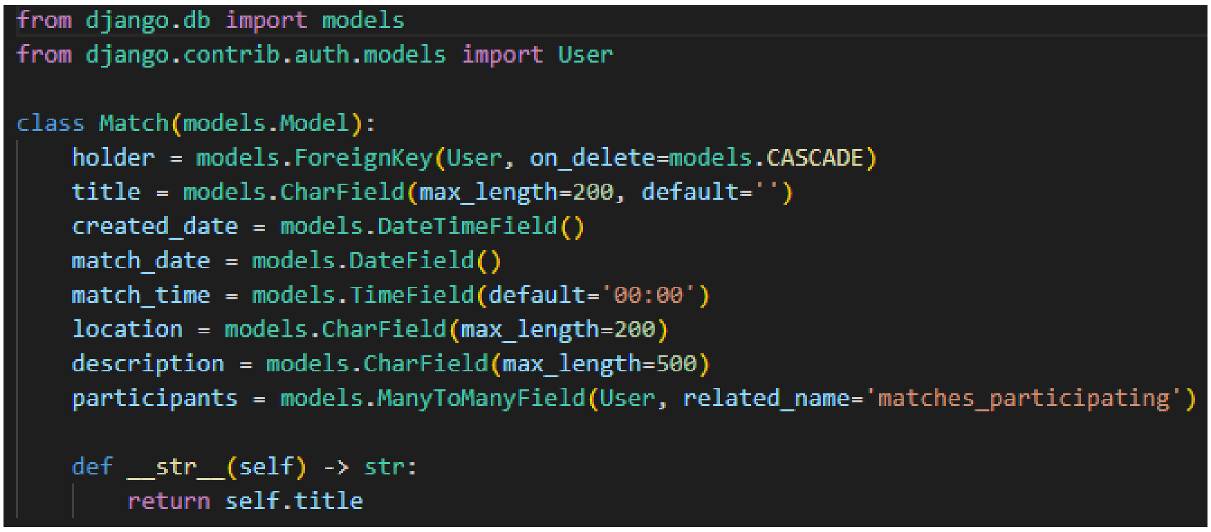
4.2.2 *match*

Match app is in charge of the match created by the users. Below are the files that are directly related to the *match* app.

```
└ match/
    └ admin.py
    └ forms.py
    └ models.py
```

```
      └ urls.py
      └ views.py
└ templates/
      └ match/
          └ create.html
          └ detail.html
```

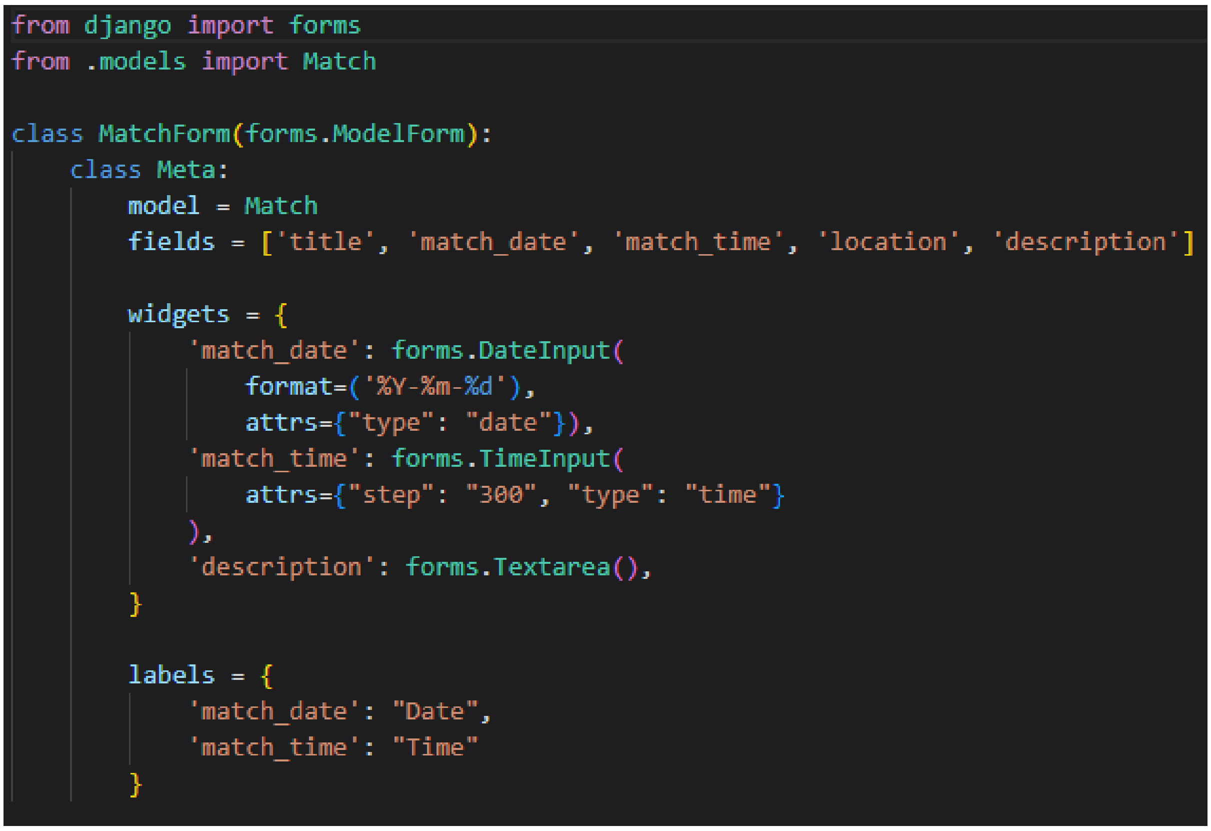
Let's look in to the *match* app by starting with the *match* model.



```python
from django.db import models
from django.contrib.auth.models import User

class Match(models.Model):
    holder = models.ForeignKey(User, on_delete=models.CASCADE)
    title = models.CharField(max_length=200, default='')
    created_date = models.DateTimeField()
    match_date = models.DateField()
    match_time = models.TimeField(default='00:00')
    location = models.CharField(max_length=200)
    description = models.CharField(max_length=500)
    participants = models.ManyToManyField(User, related_name='matches_participating')

    def __str__(self) -> str:
        return self.title
```

Figure 36. match/models.py

Data type for most fields were quite obvious apart from *participants*. This is a field for all the participants of the match, and so the first data type that came to my mind is a list. However, from the official Django document, I found *ManyToManyField* which looked perfect for the features like this. It was easier to edit the list if necessary, and most importantly using a list to store the participants requires to manually manage the list of participants which can lead to errors.



```python
from django import forms
from .models import Match

class MatchForm(forms.ModelForm):
    class Meta:
        model = Match
        fields = ['title', 'match_date', 'match_time', 'location', 'description']

        widgets = {
            'match_date': forms.DateInput(
                format=('%Y-%m-%d'),
                attrs={"type": "date"}),
            'match_time': forms.TimeInput(
                attrs={"step": "300", "type": "time"}
            ),
            'description': forms.Textarea(),
        }

        labels = {
            'match_date': "Date",
            'match_time': "Time"
        }
```

Figure 37. match/forms.py

The *MatchForm* inherit the Django *ModelForm* which consists of most basic features for Django forms. In Meta class, we define all the fields. A notable thing in figure 37 is the widgets. Widgets are UI element that is passed on to the html form to be rendered. In this example, I manually set the format of the match date. For match time, I set step as 300, which means the user will be able to select the time frame between 5 minutes. But found out, for some reason, the maximum step possible is 60, which is a minute. (Figure 38)

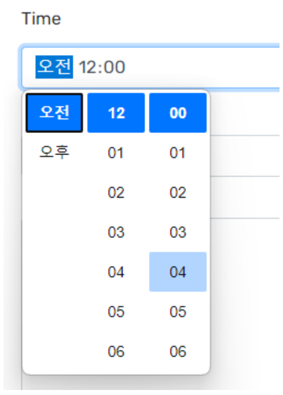


Figure 38. Setting time in match making screen

I tried to find a way to fix this as it is uncomfortable for user to scroll down all the way to 30 if the match is happening at half past. However, it seems like there's no way of doing it with *Django TimeInput* at the moment.

Next, we have description with text input area as *textarea*, which can help informing users that this field can be longer than other inputs. The labels part simply change the default name of the fields. As *match_date* and *match_time* does not necessarily look user friendly, thus I changed it to *Date* and *Time* respectively.

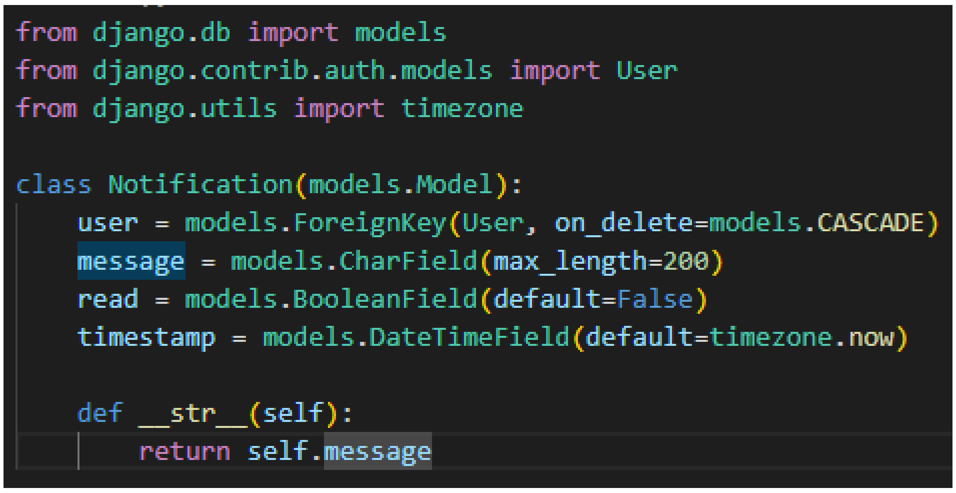
The *match* views do not contain anything noteworthy except the join and leave views. The join view simply takes the request and *match_id* and add the user to the participants list of the match and shows a message whether the user has successfully joined the match. I'm going into more detail on this later on with notification feature.

4.2.3 *Notifc*

Notifc app is in charge of the notification feature of the software. Below are the files that are directly related to the app.

```
└ futforall/
    └ consumers.py
    └ routing.py
└ notifc/
    └ urls.py
    └ views.py
    └ models.py
 └ templates/
    └ notifc/
      └ index.html
```

The notification model is relatively simple.

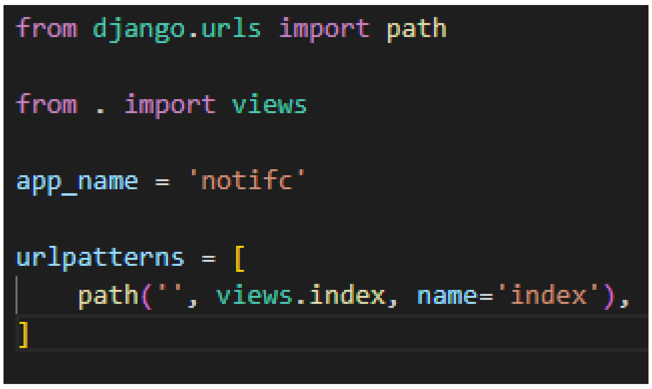


```python
from django.db import models
from django.contrib.auth.models import User
from django.utils import timezone

class Notification(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    message = models.CharField(max_length=200)
    read = models.BooleanField(default=False)
    timestamp = models.DateTimeField(default=timezone.now)

    def __str__(self):
        return self.message
```

Figure 39. notifc/models.py

Notification model has four fields. *user*, Django default model, which the notification is for.

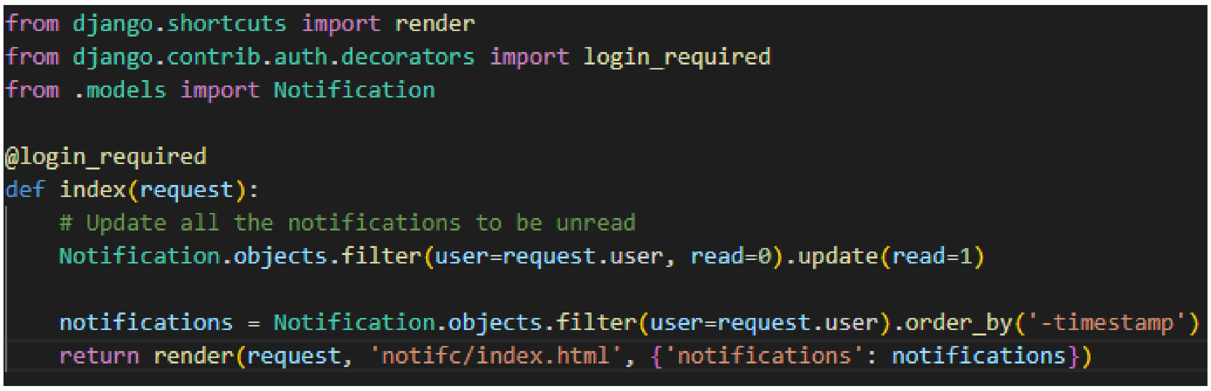
message, a string, which is the actual message of the notification. *read*, a *boolean* field, which is true if the user has read it, false otherwise. Finally, *timestamp*, a *datetimefield*, which is the date and time the notification is created. The last part of the *models.py* is a method that is called when the object needs to represented as a string. It simply returns the message attribute of the model. This is to distinguish from other notifications in the process of testing if the feature is working as how it is intended.



```python
from django.urls import path

from . import views

app_name = 'notifc'

urlpatterns = [
    path('', views.index, name='index'),
]
```

Figure 40. notifc/urls.py

There's only one url pattern which is connected to the index view of the *notifc* app. Below, figure 41, shows the index view of the *notifc* app.

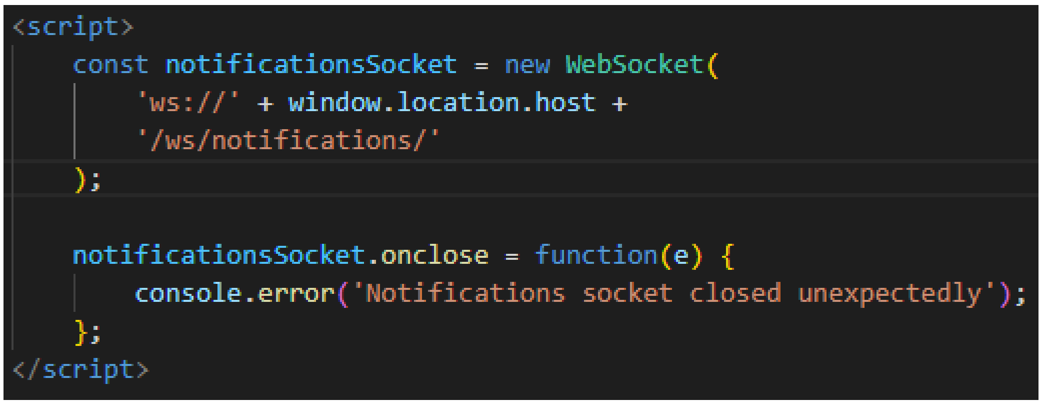


```python
from django.shortcuts import render
from django.contrib.auth.decorators import login_required
from .models import Notification

@login_required
def index(request):
    # Update all the notifications to be unread
    Notification.objects.filter(user=request.user, read=0).update(read=1)

    notifications = Notification.objects.filter(user=request.user).order_by('-timestamp')
    return render(request, 'notifc/index.html', {'notifications': notifications})
```

Figure 41. notifc/views.py

The index view is for the notification page which simply lists all the notification for the user to check. It first fetches the notifications in the database and filter the ones has user field as the user who is requesting and update all the unread notifications to read as the user is now going to read it. Then now we store all the notifications from the database that is for the user and order it by '-*timestamp*' which means it will be ordered from the recent notifications to the old ones. Then we render the page with *notifc/index.html* and pass the store notification list as context. It might seem odd, as the notification should work in real-time, and here, it is fetching the notifications from the database. Let's look into this.

When user is connected to the website, it first establishes the WebSocket connection to send and hear, if anything is being sent from the other end.

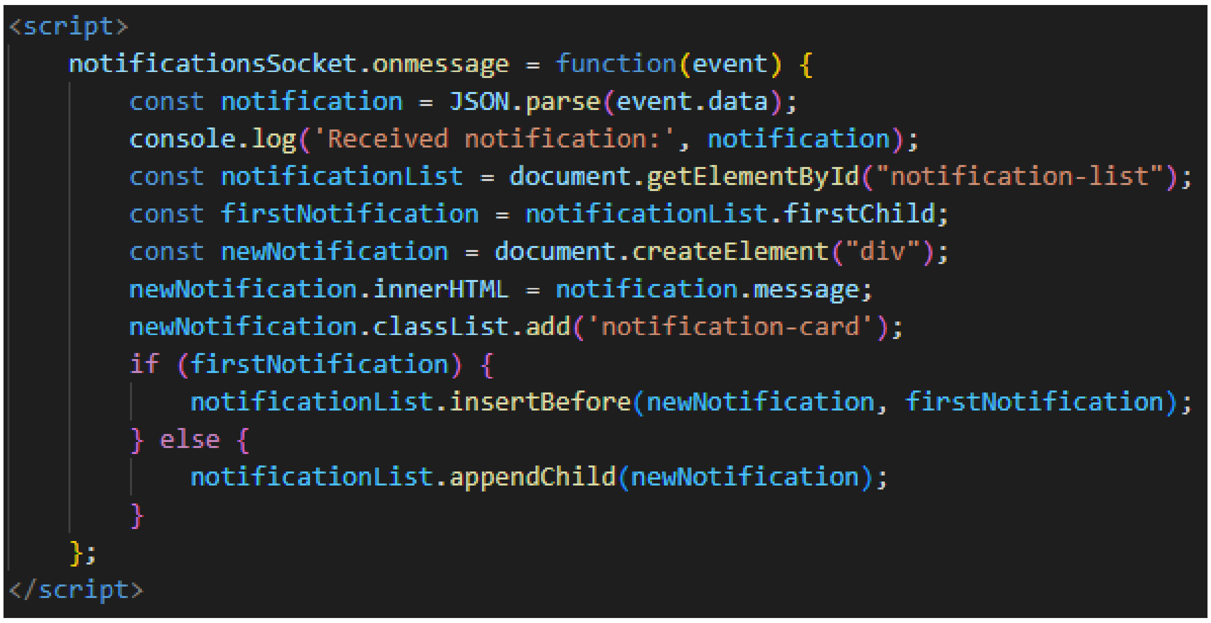


```
<script>
    const notificationsSocket = new WebSocket(
        'ws://' + window.location.host +
        '/ws/notifications/'
    );

    notificationsSocket.onclose = function(e) {
        console.error('Notifications socket closed unexpectedly');
    };
</script>
```

Figure 42. snippet from templates/base.html

The upper part of the script in figure 42 initializes a WebSocket object called *notificationSocket*. It is created by passing the url to its constructor, which in this case is the url of WebSocket server running on the current page. The lower part is an *onclose* event handler attached to *notificationSocket*, which shows an error message on the console if the connection is closed unexpectedly.

When the WebSocket connection is established, different pages have different script so it acts accordingly.

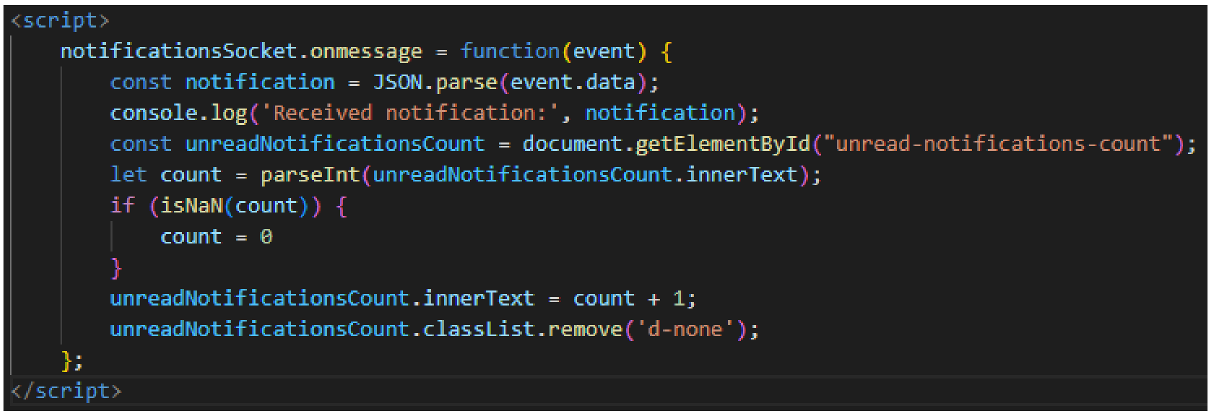


```
<script>
    notificationsSocket.onmessage = function(event) {
        const notification = JSON.parse(event.data);
        console.log('Received notification:', notification);
        const notificationList = document.getElementById("notification-list");
        const firstNotification = notificationList.firstChild;
        const newNotification = document.createElement("div");
        newNotification.innerHTML = notification.message;
        newNotification.classList.add('notification-card');
        if (firstNotification) {
            notificationList.insertBefore(newNotification, firstNotification);
        } else {
            notificationList.appendChild(newNotification);
        }
    };
</script>
```

Figure 43. snippet from templates/notifc/index.html

In the case of notification page itself, the *onmessage* handler is called whenever the WebSocket receives a message from the server. Because the notification page has to show the old notifications as well as the new ones, we have to somehow call the notifications saved in the database. But we cannot save the received notifications real-time to the database and call the notifications from the database real time, as this is extremely inefficient and this is exactly why we’re using WebSocket. The solution is to call the old notifications from the database once when the user connects to the page, and update the notification list with the notifications received via WebSocket. The newly received notifications will be saved to the database, but it won't be added real-time to the list from the database as we call from the database only once when the user is first connected to the page. Instead, we listen to the newly received notifications from the WebSocket and append the list. When the page is refreshed, the newly appended notifications will be gone, but as it is saved in the database, it will show the identical outcome.

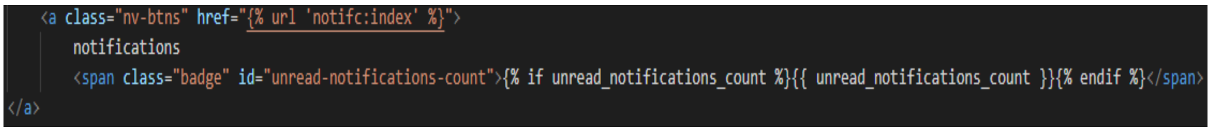
The *onmessage* handler in figure 43 first parse the *JSON* object and creates a new *div* element and save the message of the received message to *newNotification* which is wrapped with <*div*>. then the class *notification-card* is applied to give the same css style as the old notifications that are already displayed in the page. *firstNotification* is simply the first notification in the list. The if statement check if the list is empty by checking if *firstNotification* is empty and if it is not empty, add the new notification at the top of the list, and when it is empty, simply add the new notification to the list.



```
<script>
    notificationsSocket.onmessage = function(event) {
        const notification = JSON.parse(event.data);
        console.log('Received notification:', notification);
        const unreadNotificationsCount = document.getElementById("unread-notifications-count");
        let count = parseInt(unreadNotificationsCount.innerText);
        if (isNaN(count)) {
            count = 0
        }
        unreadNotificationsCount.innerText = count + 1;
        unreadNotificationsCount.classList.remove('d-none');
    };
</script>
```

Figure 44. snippet from templates/index.html

Figure 44 is a script snippet from the *index.html* which is the template for the homepage. In the homepage, we don't want to display all the notifications like notification page, but instead we want to inform the user if there's a new notification or not. The software does this by showing a badge of count of an unread notification on the notification button.



```
    <a class="nv-btns" href="{% url 'notif:index' %}">
        notifications
        <span class="badge" id="unread-notifications-count">{% if unread_notifications_count %}{{ unread_notifications_count }}{% endif %}</span>
</a>
```

Figure 45. snippet from templates/index.html

The badge in figure 45 has an if statement that check if there's an unread notification and show/hide accordingly. The *onmessage* handler in figure 44 first parse the *JSON* object. And then we fetch the unread notifications count which only counts the unread notifications from the database. In other words, if there's any unread notifications from this list, it means they are the newly received notifications when the user was not online. Then we save the count as an integer. Next it checks whether the count is a number or not. This is because the saved count of unread notifications count is null when there are no notifications as in figure

45, it only passes the *unread_notifications_count* only when it if is above 0. So, in the case of 0 unread notification, nothing is saved in constant *unreadNotificationsCount* in figure 44. *parseInt*-ing nothing will return *NaN* which literally means not a number. Thus, we have to make sure that the in the case of 0 notifications, the count is converted to 0 instead of *NaN*. After that we update the count of the notifications count. At last, we remove the *d-none* class from the *unreadNotificationsCount* element which is a *Bootstrap* class used for hiding an element.

4.3 Testing

In this project, testing was conducted in two different approaches. This part will briefly cover the process of each of those approaches and show an example each to better explain how the methods were conducted in this project.

4.3.1 Unit testing

Unit testing means testing the small functional units of the project individually to check if they are performing the way they are expected to behave by executing all the possible paths.[19] An example of testing conducted is match creation test. The match creation test is conducted by creating match with valid inputs and verifying if the match was successfully created. This can be done by viewing the Django built-in admin page. (Figure 46)

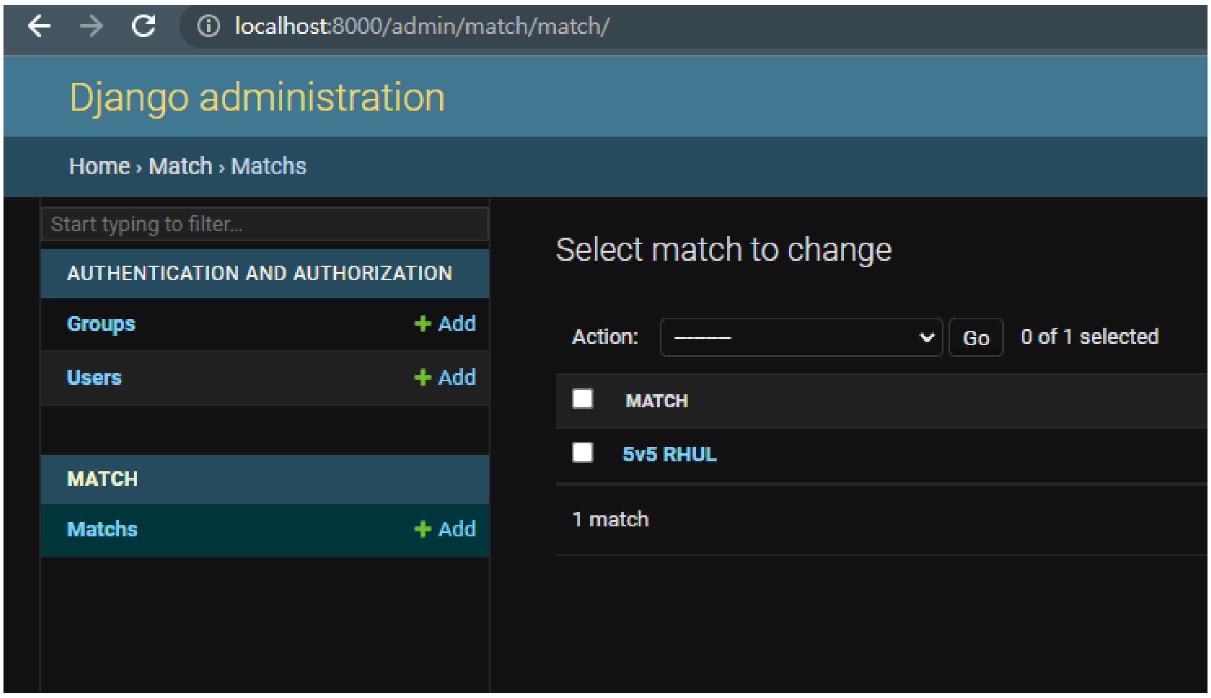
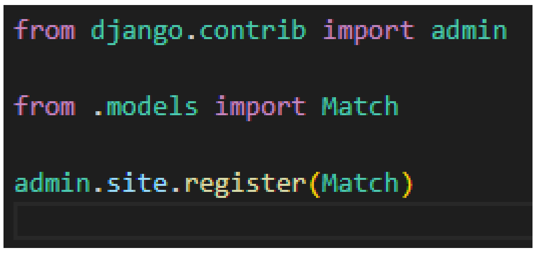

localhost:8000/admin/match/match/
Django administration
Home › Match › Matchs
Start typing to filter...
AUTHENTICATION AND AUTHORIZATION
Groups + Add
Users + Add
MATCH
Matchs + Add
Select match to change
Action: --------- Go 0 of 1 selected
MATCH
5v5 RHUL
1 match

Figure 46. Django built-in admin page

The *match* model is not added to the admin site by default.



```
from django.contrib import admin

from .models import Match

admin.site.register(Match)
```

Figure 47. match/admin.py

Using the register method in admin library of *Django*, we can simply register the *Match* model to the admin site, which allow the administrator to easily view, create, edit and delete Match objects through the admin interface.

When the testing is done the match is deleted and the other paths of match creation feature is tested again with various invalid inputs in different fields, and check if an appropriate error message is displayed.

4.3.2 Integration testing

Integration testing examines if the individual components developed and tested separately, interact as expected when they are combined to form a part of larger system.[20] This means that essentially the unit testing is done before the integration testing. An example of an integration testing for a notification feature integrated with the account features. The testing would follow the procedure below.

1. Create a test user account
2. Log in to the test user account
3. Log in to the admin account in another incognito window so that we have both accounts logged in
4. Create a match using the admin account
5. Browse to the match created with the test account
6. Join the match by clicking on join button using the test account
7. Check the web browser console of admin account and check if the notification object received is working as expected
8. Browse to the notification page and check if the notification is received and displaying correctly using admin account. (test has joined match xxx!)
9. Go to the admin page with the admin account
10. Delete the test user account

11. Check if the notifications received by the test account are deleted

This is an example of a path, but there are many different paths by reversing the role so the testing account create the match and so on. Debugging in the integration testing is more complicated than the unit testing as many different features are involved in the testing. When a test is conducted and if it didn't work as expected, I often performed testing with different paths before doing the debugging to gain more information and see if the outcome is created by specific situation.

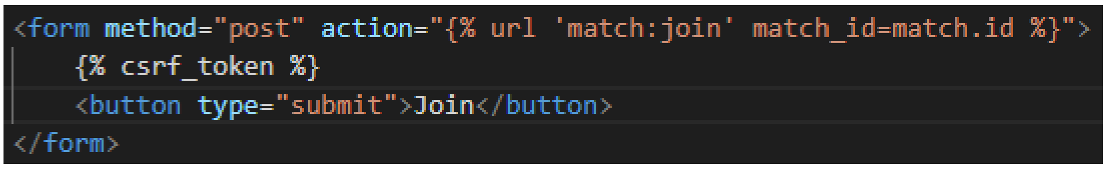
This is another example of integration testing conducted where I could identify error and fix it accordingly.

1. Create a test user account
2. Log in to the test user account
3. Log in to the admin account in another incognito window so that we have both accounts logged in
4. Create a match using the admin account
5. Log out using the admin account and disconnect from the server
6. Browse to the match created with the test account
7. Join the match by clicking on join button using the test account
8. Log back in to the admin account in another incognito window
9. Browse to the notification page and check if the notification is received and displaying correctly using admin account. (Test has joined match xxx!)
10. Go to the admin page with the admin account
11. Delete the test user account
12. Check if the notifications received by the test account are deleted

This testing looks almost identical to the previous testing but in this testing, we log out the admin account before the notification is sent. This is to test if the notification is correctly received when a user is not connected to the server.

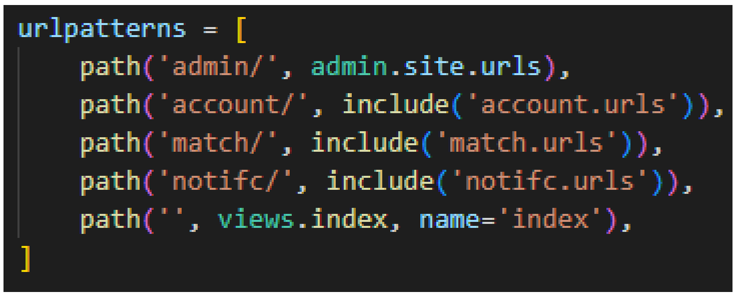
Turned out the testing has failed and the admin account did not receive any notification about the test account has joined the match created. I tried to identify the problem by printing the log message in the console in all the process of the system goes through. When the test account joins the match, below is the process of the system in terms of the file in the project.

1. *templates/match/detail.html*



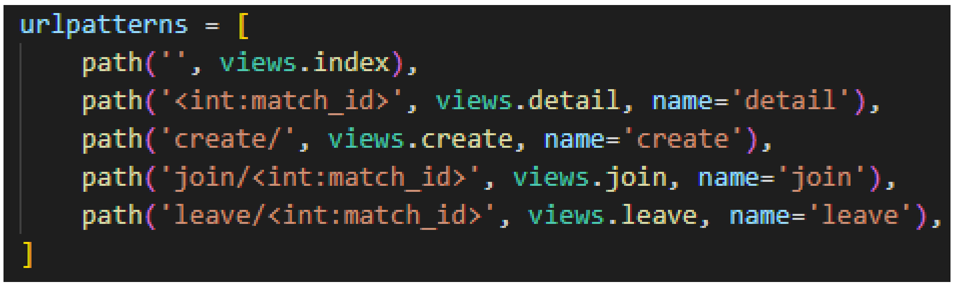
```html
<form method="post" action="{% url 'match:join' match_id=match.id %}">
    {% csrf_token %}
    <button type="submit">Join</button>
</form>
```

2. *futforall/urls.py*



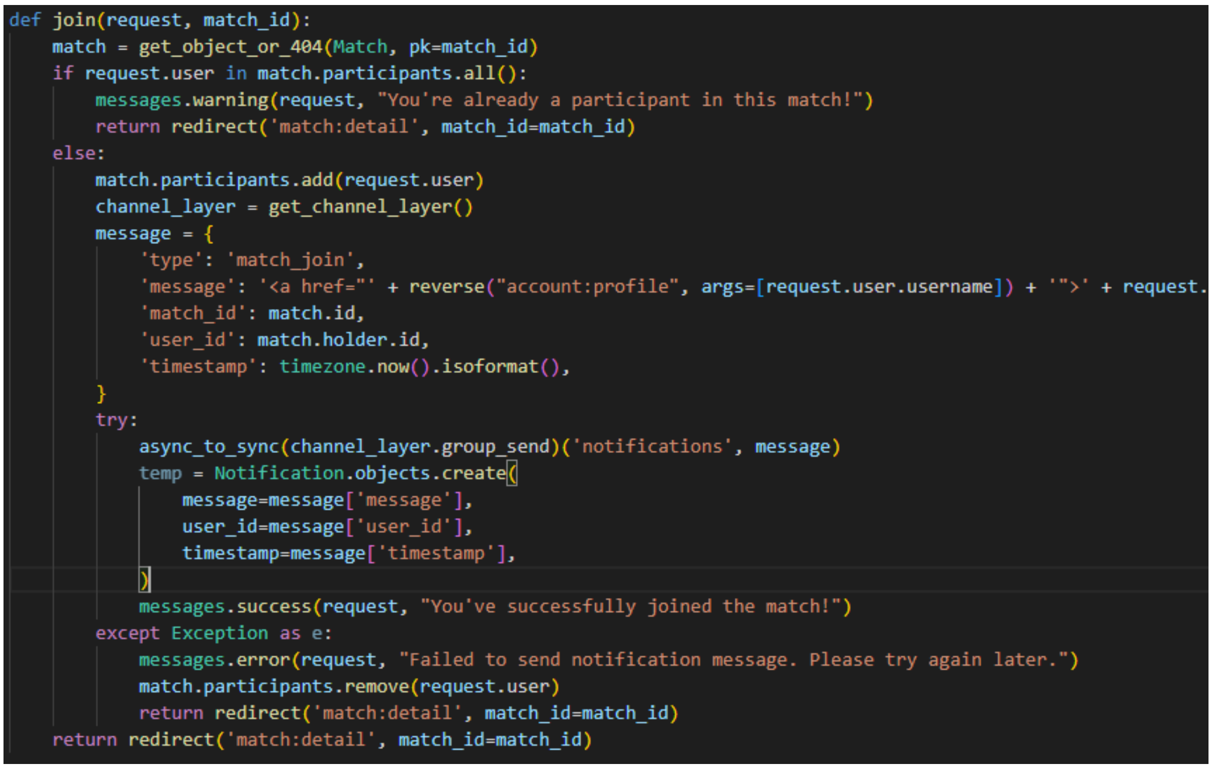
```python
urlpatterns = [
    path('admin/', admin.site.urls),
    path('account/', include('account.urls')),
    path('match/', include('match.urls')),
    path('notifc/', include('notifc.urls')),
    path('', views.index, name='index'),
]
```

3. *match/urls.py*



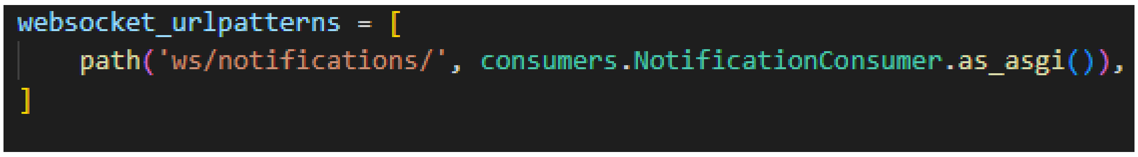
```python
urlpatterns = [
    path('', views.index),
    path('<int:match_id>', views.detail, name='detail'),
    path('create/', views.create, name='create'),
    path('join/<int:match_id>', views.join, name='join'),
    path('leave/<int:match_id>', views.leave, name='leave'),
]
```

4. *match/views.py*



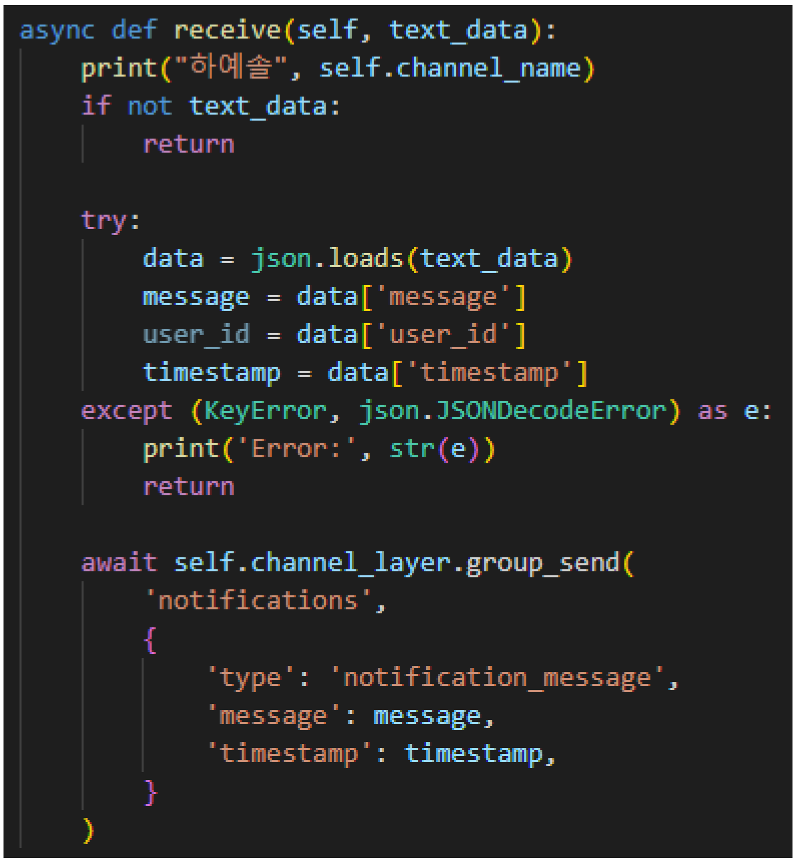
```python
def join(request, match_id):
    match = get_object_or_404(Match, pk=match_id)
    if request.user in match.participants.all():
        messages.warning(request, "You're already a participant in this match!")
        return redirect('match:detail', match_id=match_id)
    else:
        match.participants.add(request.user)
        channel_layer = get_channel_layer()
        message = {
            'type': 'match_join',
            'message': '<a href="' + reverse("account:profile", args=[request.user.username]) + '">' + request.
            'match_id': match.id,
            'user_id': match.holder.id,
            'timestamp': timezone.now().isoformat(),
        }
        try:
            async_to_sync(channel_layer.group_send)('notifications', message)
            temp = Notification.objects.create(
                message=message['message'],
                user_id=message['user_id'],
                timestamp=message['timestamp'],
            )
            messages.success(request, "You've successfully joined the match!")
        except Exception as e:
            messages.error(request, "Failed to send notification message. Please try again later.")
            match.participants.remove(request.user)
            return redirect('match:detail', match_id=match_id)
    return redirect('match:detail', match_id=match_id)
```

5. *futforall/routing.py*



```python
websocket_urlpatterns = [
    path('ws/notifications/', consumers.NotificationConsumer.as_asgi()),
]
```

6. *futforall/consumers.py*



```python
async def receive(self, text_data):
    print("하예솔", self.channel_name)
    if not text_data:
        return

    try:
        data = json.loads(text_data)
        message = data['message']
        user_id = data['user_id']
        timestamp = data['timestamp']
    except (KeyError, json.JSONDecodeError) as e:
        print('Error:', str(e))
        return

    await self.channel_layer.group_send(
        'notifications',
        {
            'type': 'notification_message',
            'message': message,
            'timestamp': timestamp,
        }
    )
```

As the previous testing was successful, I could narrow down that the problem was in either 4 or 6 stage in above. By console logging and printing on terminal in case of *py* files, I could identify that the receive function was not ran during the process, and that's a problem as previously the system was saving the notification to the database in the receive function. I could double check by browsing to the database, and indeed the notification was not saved. By performing various testing, I could identify that when the test user joined the match, it sends the message to the channel layer with group of *'notifications'* and obviously the admin account is logged out, so the admin user was not in the group. The solution was simple. Instead of saving the notification to the db in receive function, it can be done in the join function in *match/view.py*. this way, we don't need websocket connection for the notification to be saved, but also, I had to make sure when the notification is failed to send, the notification should not be saved in the database. This solution has fixed the problem, and also, I could understand how the WebSocket connections work as I was still in the process of learning.

Chapter 5: Experiences and review

This chapter elaborates on the problems I had, and review in terms of the problems and reflect how it could be done in better way.

5.1 Obstacles on learning new technologies

Learning the knowledge of the technologies used in this project proved to be a much more significant part of the entire process than I expected. Prior to this project, my experience was limited to Git and Python out of the technologies that I used. While attempting to stick to the official documentation, I faced several limitations since official documentation usually offer only minimum examples. I resorted to watching multiple tutorials and reading several documents, but at times, I felt stuck and as if I wasn't making any progress. To overcome these obstacles, I tried to approach problems by sticking to the requirements of the project and breaking them down into the smallest possible components. I even created a separate learning project for some of the more complex problems I encountered, learning each concept step by step, not worrying about the main project. At the end, everything worked out, but the amount of time I invested in learning far surpasses my expectations at the beginning of the project.

Looking back, I believe my preparation for the project was far from enough. Commencing the learning of the necessary technologies during the summer holiday would have provided me with more time and deeper understanding of the underlying technologies I used. It's worth mentioning that the project topic was changed at the last minute in September, which was an exception.

5.2 Poor Documentation

In this project, I used Sphinx to generate the project's documentation, but it is obvious that it lacks in content. There are several factors that contributed to this issue, including the software methodology approach I used, my initial oversight of the importance of documentation, and my inexperience in using Sphinx. As discussed in Chapter 3, the agile method prioritizes working software over comprehensive documentation, which led me to believe that documentation was not a priority and was therefore placed on the backburner. Even when I eventually began to generate the documentation, I found out that setting up Sphinx and learning how to use it was time-consuming and challenging. While I was the only developer of the project, documentation is an essential aspect of any project as it serves as the most basic form of communication.

To improve the quality of the project's documentation, I could have learned how to use Sphinx before the project and documented my code as I progressed, not after the software is done.

5.3 Limited Testing

Looking back now, the testing for the project could have been far better in terms of quality and quantity. There are many well-structured testing methodologies which rely less on human capabilities that I could utilize. Also, as I was the only developer, the testing was done only on my environment which can lead to different outcomes in different environments and with different testers.

To improve the testing, I could approach the project with test driven software methodologies, and even not utilizing the test-driven development methods, I could perform structured testing using testing apps.

5.4 Poor Planning

By reading the project plan I submitted at the beginning of the year, I can tell some parts of the planning were poor in terms of reality and scope. In terms of scope, the plan is not thought to details in some part that I lack some of the essential part of the project, documentation for instance. Also, the features I aimed to implement are not all implemented. While features like real-time notification are implemented, features like friend system are not implemented. This is due to my choice with limited time and resources. As the project is for academic purposes, I decided to put real-time notification on priority as it is more advanced features than other features.

This could be improved by considering the whole process in more detail, but also the lack of knowledge and deep understanding of the framework definitely contribute to the issue.

5.5 Personal Problems

This issue is less significant than other problems as this is not a structural problem. But after the Christmas break, I had some medical problems myself that I had to go to see my GP several times. This problem was crucial for the project as I am the only developer.

Chapter 6: Professional Issues

Addressing professional issues in software development is crucial for ensuring ethical and responsible use of technology that benefits society. My project deals with creating a platform, futforall, for managing and playing football matches, and there are several professional issues that should be dealt with.

Firstly, futforall is a service that helps people from different societal background to find football matches and participate. It is important to design the user interface of the platform in a way that is accessible and easy for people of different ages, sex, cultural background and more. This is a challenging task as it is limited for myself to see in the eyes of different people. One way I could tackle this problem was to perform a research on the topic and see how other well-known platforms tackle this problem and implementing well-known designs. One notable thing I did was to make sure the size of the font is not too small for people with lower vision.

Privacy is an important concern on the internet, which is an ocean of information. As futforall is a web application, it is crucial to prioritise user privacy and ensure that it is not compromised. While implementing the right technologies for security is necessary, the social networking features of the web application demand careful planning to prevent any potential privacy breaches. The social network feature of futforall is not implemented, but this is still something essential to take note on. In this project, I tried my best to utilise the built-in security features of Django like CSRF tokens.

Lastly, management of the project is also important in how the resources like time is managed during the project. This part was mainly discussed in Chapter 5.

Chapter 7: Project Information

This part of the report includes a description of how to run the web application, and the environmental requirements. Also, a link to the video that shows a deployment will be included.

7.1 Instructions

1. Python (above version 3.10) should be installed and added to the PATH environmental variable

2. Docker desktop should be installed and it should be running on the background

3. In Windows Command Prompt, browse to the project directory *'...\PROJECT\futforall'*

4. run the following command to activate the virtual environment

    .\Scripts\activate.bat

5. install all the required frameworks by running the following command

    pip install -r requirements.txt

6. on the command line, run the following command

    docker run -p 6379:6379 -d redis:6

7. on the command line, run the following command

    python manage.py runserver

8. Now connect to the website with following url; *localhost:8000*

7.2 Application demo video

Below is the YouTube link to the demo video. The video has no sound but subtitles.

https://youtu.be/ld2ZkYLvFSc

Chapter 8: Diary

This part is the diary that I have written throughout the development of the project. Please take in mind the diary is informal and might not be directly related to the content of the project.

```
28/SEP/22
- created branch
- wrote abstract draft
- cloned branch to my pc
- added plan.txt
- edited plan.txt to include the abstract I've written past few days

02/OCT/22
- updated timeline of the project plan
- website structure draft

- completed timeline draft term 1

- updated abstract so it includes more about my drive to make this
website I'm
  making
- completed abstact draft

- completed timeline draft term 2
- updated abstract

- completed risk assessment draft

- completed Project Plan draft in the form of docx

04/OCT/22
- got feedback on plan draft
- updated abstract according to the feedback

- completed second draft of abstract
- completed second draft of timeline
- completed second draft of risk assessment

06/OCT/22
- final check
- corrected grammars and spelling
```

- submitted the project plan to moodle
- i can see feedback from last year. I got a F from Dave and it says "The plan is not submitted, because the
  student has taken a year off..."

- decided to go with django, I love Python.
- had a looked at django tutorials

- added a static website for testing purpose

08/OCT/22

- installed wampstack (apache)

- cgi setted up

- learned cgi & python
- now I have simple website running on Apache

09/OCT/22

- learned about XSS
- learned about package manager

- learned about html-sanitizer

09~22/OCT/22

- revised git ...
- learned about gitlab
- reverted and fixed repo

- learned about structure of django
- learned about how to read, write in django

- learned about virtual environment

- fixed issue where import is not working in the virtual environment created

- fixed issue where views.py not being able to call main.html file from the

  templates folder

23/OCT/22

- continue learning about django

- created navigation bar

- created match lobby in homepage

- added meta data to the match (just for structure use for now)

this is all frontend yet

24/OCT/22

- created match app

- create Feed Class which will be the matches shown in the homepage

- edited views to fetch feed_list from DB

- edited main.html in templates to display matches fetched from DB

- created accounts app

- created login and signup.html templates for accounts app

- basic bone structure of signup.html

29/OCT/22

- made basic frame of the right body of the homepage to display the profile of

  the user and the list of the friends of the user

- made the match add modal

- the match add modal now pops up when the user click the add match button on the navigation bar

- the match add modal now closes when the user click the close button at the
  right top corner of the modal

09/NOV/22

Came back from illness ..

- started writing project report
- project report title
- project report table of contents

14/NOV/22

Came back from assignments piled up during my illness ...

- started writing project report
- thought about structure for Part 1 (Abstract + Project specification +
  Introduction etc)
- copied and paste the abstract from the project plan to start with
- wrote a bit of project specification then wonder if this should be written
  by me or am I supposed to use the original specification so sent an email to
  the supervisor
- came back to source code and I'm lost, I don't know what's happening
- decided to go back to scratch and start without rest framework which I lack
  understanding
- managed to make it work without it (pure django) but scripts are not
  working... tons of working incoming.

16/NOV/22

- created new project to practice django from beginning
- created django app called polls
- created view for polls
- created urls.py for polls and called view
- connected polls urls.py to the main urls.py

19/NOV/22

- finished django tutorial part 2 and 3

21/NOV/22

- finished django tutorial (only neccessary parts)

22/NOV/22

- created new project for actual project
- created super user and set up admin page

- created app for login
- set urls for login and connect it to the main urls.py
- created template for login page
- created form error template

- views.py for homepage
- connected it to urls
- created homepage template

- forms.py for sign up form
- created template for signup page
- connect it to urls.py

- created match app
- match model with adequate attributes
- MATCH APP CONFIG TO SETTINGS PY (NEVER FORGET OR OTHERWISE HELL FOR ANOTHER
  HOUR)
- NVER FORGET MIGRATION AS WELL
- added match to admin page for easier management (and test)
- created match create page (just scratch really)

24/NOV/22

```
- created form for match creation

- widgets for datetime and description (textarea) (I used my own anyway, this
  was useless)

- seperated base.html into base.html which contains <head> only with style and
  base(nav).html which also contains navigation bar. (this is because some
  pages don't need nav bar)


- created profile page and template

- profile page take username in url instead of primary key. cool.

- and ui/design improvement in all pages

- my website now looks cool


- next step is to expand match meta data

- also learn how to expand builtin django user to add custom attributes

- also learn about uploading and storing images (for profile pic) (no rush
  this is for social media feature for term 2)


- time to write report (least fav part)


26/NOV/22

- added Match to index view and it pass match list to index.html

- index.html now display all active matches


- ../account/admin is not working some reason, while all other user's profile
  pages displays correctly. Changed the name of the superuser to avoid this
bug


- time to write report


- completed project specification
```

27/NOV/22

- Introduction completed

- 1.1 Motivation completed

- 1.2 Aims of the project completed

- Read Evaluating web development frameworks: Django, Ruby on Rails and CakePHP by Julia Plekhanova

28/NOV/22

- Read Full-stack web development using Django REST framework and React by Joel Vainikka

(very interesting report about investigating already existing Finnish Esports League's web service which is created with Wordpress and creating a new service)

- Read Designing an MVC Model for Rapid Web Application Development by Dragos-Paul Pop, Adam Altar

- completed 1.3 comparing web development frameworks

29/NOV/22

- Chapter 2: Architecture start

- 2.1 virtual environment and installations completed

- 2.2 File structure completed

- 2.3 Features completed

- diary added to the report

17/JAN/23

- it's time to add friends feature

- fixed the placements of items in profile page

- added name to the profile page

10/FEB/23

- added friends model

- add_friend view added to account/views.py
- added url pattern for adding a friend

- friend list now displayed on the profile page
- add friend button displayed next to the name

- in order to accept friend request, we need notification feature
- so we pause with the friend feature here, and it's time to implement notification feature

- did some research and found out I don't need push notification as this is a website, it is more suitable to use notification feature where it's only visible when user is in the web app

- I'm gonna use Django Channels for web socket

- time to learn how to use Django Channels

20/FEB/23

- installed channels and setting environments

- created 'notifc' which is an app for notification feature

- My plan is to follow the official Django Channels document's tutorial which

  is a chat app using websocket.

21/FEB/23

- I'm not getting any of the official documents so am watching youtube videos and tutorials on channels

- i need to learn about websockets and async mechanics first

30?/FEB/23

- watching some easy explanation in my first language for better understanding

10/MAR/23

- deleted the whole practice project because it got all tangled as I was

  practising with many different tutorials

- created new venv and project to start fresh for learning

- followed the official channels document chat app tutorial

- it works and I'm feeling like I understand better than a month ago

- but i don't feel like I should go straight into my main project with the

  risk of ruining it

- thus my plan is to do another practice project, but this time on my own and

  also not chat app but an actual notification app

- but i don't wanna do it today so for today let me do fun designing

- completed practice for dropdown menu that I'm planning for notification

13/MAR/23

- resuming my notification practice project

- it's not as simple as I thought it would be

- struggling.

14/MAR/23

- resuming from where I left off

- for some reason it's not recognising channels and daphne that I've clearly
  installed
- trying to fix it but stackoverflow is no help rn

- for some reason, I had to install channels and daphne globally not in my
  venv. it's weird... why is it not looking it from my venv lib folder?

- resuming the practice

- it does not work, i don't know what's the problem
- let's do some other things for now

- match participants system complete

- not 100% sure what daphne does
- and redis too

- finally it's working, not correctly, but at least there's websocket
  connection established

- I'm getting the notification in the console, I can see it but it's not
  displaying??

15/MAR/23

- so the websocket is real-time thing, but it's ONLY real-time thing... why
  did I think it's gonna be stored automatically
- so that means I need to save it to the db and fetch it from db?

- but then when I need to list the notifications I need to check the new
  notifications in db real time? there's no point of async in that case... and
  also that's nonsense, performance would be so bad

- the clever way (and maybe the only way) was to fetch from db, but also put
  real time ones on the top of the list but for user it will feel like they
  are the same thing. so now it's time to save them to the db

- websockets request are sent twice??? why..? it's saved in the db twice too
  cus it's calling receive function twice every time a user join a match.

- seems like that's how it works as it's two users websocket connected to each
  other?

- the ultimate solution was to save it to the db when send the notification...
  not when a user receive it. simple but very efficient wow.

- testing this is so tiring.. I can't make match for every single time I test.

- created view for leaving a match and added a leave button

16/MAR/23

- now it's time to show red dot in the notification button when there's new
  notifications

- establish websocket connection when the user connect to the website, not
  just in the notification page

- some struggle but it works now

- documentation for django
- installed sphinx
- learning how to use it

- created documentation to see if it works
- it works but so many unwanted documents
- gotta tidy up in the future...

- it's time to begin writing reports

- following up on the feedbacks received for interim reports

- chapter 3 software dev methodology completed
- tried to write more so that I can get some feedback for supervisor meeting

  tomorrow but it's not possible I need sleep

17/MAR/23

- supervisor meeting

1. I don't lose mark for not implementing every feature I said in the project

plan, I need to justify it though

2. Assume the markers are using windows as I used windows

3. gotta mention about testing even if it's more informal type of testing

4. readme file of installations and all, but also highlights of where to look

into

- working on reports

18/MAR/23

- chapter 3 adding on

19/MAR/23

- Chapter 4 start

20/MAR/23

- planned to send it yesterday but this is not enough
- chapter 4 midway

- sent it to Mr. Lange

22/MAR/23

- received feedbacks
- revising chapter 2 based on the feedback

- chapter 2 updated with new diagram and updated the old one too
- changed to order to be more sensible

23/MAR/23

- chapter 4 continued

27/MAR/23

- chapter 4 done
- planning the rest of the report

- it's gonna include Experiences and review and professional issues

30/MAR/23

- chapter 5 done
- chapter 6 done

- demo video created

- chapter 7 in progress

- chapter 8 is diary so this is going to be the last diary, bye

References

[1] A short history of the Web. https://home.cern/science/computing/birth-web/short-history-web

[2] Grech V. Publishing on the WWW. Part 5 - A brief history of the Internet and the World Wide Web. (2001). https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3232505/

[3] Gazdecki, Andrew. “Why Progressive Web Apps Will Replace Native Mobile Apps.” (2018). https://www.forbes.com/sites/forbestechcouncil/2018/03/09/why-progressive-web-apps-will-replace-native-mobile-apps/?sh=20cec6c42112

[4] Thakur, Parbat. "Evaluation and implementation of progressive web application." (2018). 1-2.

[5] Plekhanova, Julia. "Evaluating web development frameworks: Django, Ruby on Rails and CakePHP." Institute for Business and Information Technology 20 (2009): 2009. 4.

[6] Django documentation. November 2022. https://docs.djangoproject.com/en/4.1/topics/templates/

[7] Django Tutorial Part 4: Django admin site. November 2022. https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Admin_site

[8] Oracle official website. November 2022. https://www.oracle.com/uk/database/what-is-database/

[9][10] Moniruzzaman, A. B. M., and Syed Akhter Hossain. "Nosql database: New era of databases for big data analytics-classification, characteristics and comparison." *arXiv preprint arXiv:1307.0191* (2013). 1.

[11] Django documentation. November 2022. https://docs.djangoproject.com/en/4.1/intro/tutorial01/

[12] Django documentation, working with forms. November 2022. https://docs.djangoproject.com/en/4.1/topics/forms/

[13] Pop, Dragos-Paul, and Adam Altar. "Designing an MVC model for rapid web application development." Procedia Engineering 69 (2014): 1172-1179.

[14] Abrahamsson, Pekka, et al. "Agile software development methods: Review and analysis." *arXiv preprint arXiv:1709.08439* (2017). 9

[15] Abrahamsson, Pekka, et al. "Agile software development methods: Review and analysis." *arXiv preprint arXiv:1709.08439* (2017). 11

[16][17] Abrahamsson, Pekka, et al. "Agile software development methods: Review and analysis." *arXiv preprint arXiv:1709.08439* (2017). 13

[18] Django documentation. November 2022. https://docs.djangoproject.com/en/4.1/intro/tutorial01/

[19] Brar, Hanmeet Kaur, and Puneet Jai Kaur. "Differentiating integration testing and unit testing." *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*. IEEE, 2015. 796

[20] Brar, Hanmeet Kaur, and Puneet Jai Kaur. "Differentiating integration testing and unit testing." *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*. IEEE, 2015. 797