

Find My Team (FMT)

Justin Perry Doyle

N00211756

Supervisor: Stefan Paz Berrios

Second Reader: John Montayne

Year 4 2024/25

DL836 BSc (Hons) in Creative Computing

Abstract

Find My Team (which will be referred to as “FMT” throughout this report) is a full stack web application, that aims to provide users with a way to connect with other players (specifically of the game League of Legends or “LoL”) and play together competitively. FMT does this by allowing users to connect their FMT account to their LoL/RIOT account (RIOT are the developers of LoL) and inherit statistics and rank that they achieve in game into FMT. These statistics are then displayed on a user card in a similar fashion to that of dating applications like Tinder or Hinge, where users may select which players to play or “match” with by liking that persons’ profile. The steps involved in developing FMT include designing a back end, front-end, and UI interface, doing research in all these areas and putting that into practice to create a functioning web application for users. Testing was carried out before, during and after implementation. The market research showed a clear desire for FMT in the landscape and that people approached FMT with familiarity after most having had experience with similar applications although centred around dating rather than gaming. To expand FMT, adding new games into the application would be the first place to start, and finding ways to integrate data analytics and machine learning into the search algorithms may be helpful and useful.

Acknowledgements

Many thanks to my supervisor Stefan Paz Berrios, for his unwavering availability and enthusiasm, motivating me to be the best version of myself and keep striving for better throughout the project.

My second reader John Montayne, for being insightful and helpful, gave me important guidance at pivotal times during the project.

All my lecturers, without their learning this project would not have been possible.

IADT and the department of Technology & Psychology, for facilitating my learning over the past four years, allowing me to become a more educated and better version of myself.

My family, for supporting me throughout my academic studies, and who kept faith in me to succeed at all times.

**The incorporation of material without formal and proper acknowledgement (even with no deliberate intent to cheat) can constitute plagiarism.**

If you have received significant help with a solution from one or more colleagues, you should document this in your submitted work and if you have any doubt as to what level of discussion/collaboration is acceptable, you should consult your lecturer or the Course Director.

**WARNING**: Take care when discarding program listings lest they be copied by someone else, which may well bring you under suspicion. Do not to leave copies of your own files on a hard disk where they can be accessed by other. Be aware that removable media, used to transfer work, may also be removed and/or copied by others if left unattended.

Plagiarism is considered to be an act of fraudulence and an offence against Institute discipline.

Alleged plagiarism will be investigated and dealt with appropriately by the Institute. Please refer to the Institute Handbook for further details of penalties.

**The following is an extract from the B.Sc. in Creative Computing (Hons) course handbook. Please read carefully and sign the declaration below**

*Collusion may be defined as more than one person working on an individual assessment. This would include jointly developed solutions as well as one individual giving a solution to another who then makes some changes and hands it up as their own work.*

|  |
| --- |
| **DECLARATION**:  I am aware of the Institute’s policy on plagiarism and certify that this thesis is my own work.  Student : Justin Perry Doyle  Signed Justin Perry Doyle |

Failure to complete and submit this form may lead to an investigation into your work.

Table of Contents

[1 Introduction 1](#_Toc196943135)

[1.1 Technologies 1](#_Toc196943136)

[1.2 Tools 1](#_Toc196943137)

[2 Research 2](#_Toc196943138)

[2.1 Introduction 2](#_Toc196943139)

[2.2 AI in Recommender Systems 3](#_Toc196943140)

[2.3 Types of Recommender Systems 6](#_Toc196943141)

[2.4 Statistical Models in Recommender Systems 9](#_Toc196943142)

[2.5 Uses of Recommender Systems 13](#_Toc196943143)

[3 Requirements 15](#_Toc196943144)

[3.1 Introduction 15](#_Toc196943145)

[3.2 Requirements gathering 16](#_Toc196943146)

[3.2.1 Similar applications 16](#_Toc196943147)

[3.2.2 Interviews 17](#_Toc196943148)

[3.3 Requirements modelling 21](#_Toc196943149)

[3.3.1 Personas 21](#_Toc196943150)

[3.3.2 Functional requirements 21](#_Toc196943151)

[3.3.3 Non-functional requirements 23](#_Toc196943152)

[3.3.4 Empathy Maps 23](#_Toc196943153)

[3.3.5 Journey Maps 25](#_Toc196943154)

[3.3.6 User Scenarios 27](#_Toc196943155)

[3.4 Feasibility 28](#_Toc196943156)

[3.5 Conclusion 29](#_Toc196943157)

[4 Design 30](#_Toc196943158)

[4.1 Introduction 30](#_Toc196943159)

[4.2 Program Design 30](#_Toc196943160)

[4.2.1 Technologies 30](#_Toc196943161)

[4.2.2 Structure of Python Flask & React JavaScript (2 pages) 31](#_Toc196943162)

[4.2.3 Design Patterns 33](#_Toc196943163)

[4.2.4 Application architecture 33](#_Toc196943164)

[4.2.5 Database design 33](#_Toc196943165)

[4.3 User interface design 35](#_Toc196943166)

[4.3.1 Wireframes 36](#_Toc196943167)

[4.3.2 User Flow Diagram 40](#_Toc196943168)

[4.3.3 Style guide 42](#_Toc196943169)

[4.4 Conclusion 42](#_Toc196943170)

[5 Implementation 43](#_Toc196943171)

[5.1 Introduction 43](#_Toc196943172)

[5.2 Development environment 43](#_Toc196943173)

[6 Database Development & Implementation 44](#_Toc196943174)

[6.1 SQLLite 44](#_Toc196943175)

[6.2 PostgreSQL 45](#_Toc196943176)

[7 Back End Development & Implementation 46](#_Toc196943177)

[7.1 Introduction to Python Flask 46](#_Toc196943178)

[7.2 User 46](#_Toc196943179)

[7.2.1 Model 46](#_Toc196943180)

[7.2.2 Controller & Routes 49](#_Toc196943181)

[7.3 Group & UserGroup 58](#_Toc196943182)

[7.3.1 Model 58](#_Toc196943183)

[7.3.2 Controller & Routes 58](#_Toc196943184)

[7.4 Matches 61](#_Toc196943185)

[7.4.1 Model 61](#_Toc196943186)

[7.4.2 Controller & Routes 61](#_Toc196943187)

[7.5 Stats 64](#_Toc196943188)

[7.5.1 Model 64](#_Toc196943189)

[8 Front End & UI Development & Implementation 65](#_Toc196943190)

[8.1 Overview 66](#_Toc196943191)

[8.2 Contexts, Configs, Router and Navigation 66](#_Toc196943192)

[8.3 Register & Login 70](#_Toc196943193)

[8.4 Home Page 77](#_Toc196943194)

[8.5 Social Page 88](#_Toc196943195)

[8.6 Profile Display 100](#_Toc196943196)

[8.7 Helper Page 101](#_Toc196943197)

[9 Testing 102](#_Toc196943198)

[9.1 Introduction 102](#_Toc196943199)

[9.2 Functional Testing 102](#_Toc196943200)

[9.2.1 CRUD + Auth 103](#_Toc196943201)

[9.2.2 API Ingestion & Connection 104](#_Toc196943202)

[9.2.3 Groups & Matches 104](#_Toc196943203)

[9.2.4 Discussion of Functional Testing Results 105](#_Toc196943204)

[9.3 User Testing 106](#_Toc196943205)

[9.4 Conclusion 108](#_Toc196943206)

[10 Project Management 109](#_Toc196943207)

[10.1 Introduction 109](#_Toc196943208)

[10.2 Project Phases 110](#_Toc196943209)

[10.2.1 Proposal 110](#_Toc196943210)

[10.2.2 Requirements 110](#_Toc196943211)

[10.2.3 Design 111](#_Toc196943212)

[10.2.4 Implementation 111](#_Toc196943213)

[10.2.5 Testing 111](#_Toc196943214)

[10.3 SCRUM Methodology 112](#_Toc196943215)

[10.4 Project Management Tools 112](#_Toc196943216)

[10.4.1 GitHub 113](#_Toc196943217)

[10.4.2 Miro 113](#_Toc196943218)

[10.4.3 Teams 113](#_Toc196943219)

[10.5 Reflection 113](#_Toc196943220)

[10.5.1 Your views on the project 113](#_Toc196943221)

[10.5.2 Completing a large software development project 113](#_Toc196943222)

[10.5.3 Working with a supervisor 114](#_Toc196943223)

[10.5.4 Technical skills 114](#_Toc196943224)

[10.5.5 Further competencies and skills 114](#_Toc196943225)

[10.6 Conclusion 114](#_Toc196943226)

[11 Conclusion 116](#_Toc196943227)

[References 118](#_Toc196943228)

# Introduction

The aim of this project is to supply users with a web application, designed to allow them to connect with other users who play the game League of Legends. This is done by allowing them to connect their LoL/RIOT account to FMT, inheriting their in-game statistics, which will be used in user cards in a similar format to that of dating apps to allow users to like or match with one another based on filters they decide upon (rank, score, kills etc.).

While statistical tracking applications like that of Tracker Network exist, and matching applications like those of Tinder & Hinge also exist, no app that combines these two mediums is currently available in market. This gap will be filled by FMT where users will be able to connect through data driven insight and a seamless user interface.

## Technologies

The technologies to be used in the creation of FindMyTeam:

* SQLite (Development DB)
* PostgreSQL (Deployment DB)
* Python Flask
* ReactJS
* RIOT API
* Insomnia
* Tailwind/Shadcn
* Discord API

## Tools

The tools to be used in the creation of FindMyTeam:

* Github
* Miro
* Teams
* VSCode

# Research

The research section for this thesis was completed on recommender systems, the AI inside of them, how these systems are used in apps made to bring people together & the issues found with these systems in said apps, this research was then used to inform the development of FindMyTeam, and how a system could be created in its own context, to escape these pitfalls.

## Introduction

Abstract

Recommender systems are becoming increasingly prevalent in many applications and systems designed and used by companies and individuals alike. These systems are designed to help consumers and users have relevant information or items displayed to them based on their and other similar users preferences, this lessens the time users spend searching for what they are looking for using personalization and helps companies maximise profit.

In this paper discussed will be the basis of recommender systems, how they work and are used, deep learning within recommender systems and the use of AI within and in conjunction with recommender systems, the statistical models used by these AI learning models to produce accurate and reliable recommendations to users and these systems’ uses in business, and some of their shortcomings when used or designed incorrectly.

Introduction

Recommender systems are tools that filter information or products to a user or consumer, by using data about that user and/or similar users to create exciting content in a personalized manner (Schafer et al., 2001). Today a user can instantly access descriptions, advertisements, comments, and reviews about an almost limitless amount of products and content. Recommender systems are designed to make these things more relevant to the user and more impactful for the companies who run these platforms and own these products (Batmaz et al., 2019).

In a recommender system, items or content are recommended to users, this item/content is stored usually in a list form based on a prediction of the users affinity for said item or content. This is done by analysing the users past preferences (content based filtering) , by analysing the patterns of similar/like-minded users (collaborative filtering) or a combination of both (hybrid filtering), this suggestion is then shown to the user with the hope that the user is more likely to be interested in this item/content (Batmaz et al., 2019).

AI techniques have started to be applied to recommender systems more recently, helping to enhance the experience of the user and heighten user and in turn provider satisfaction. It enables higher quality suggestions to be shown to the user, enabling the users experience to be more efficient, more worthwhile and more enjoyable, hence driving traffic and profit for the providers of these systems (Zhang et al., 2021).

## AI in Recommender Systems

Convolutional Neural Networks

Convolutional neural networks are an incredibly important part of the field of AI and deep learning. A CNN is a type of feedforward neural network that extracts features from data similar to feature extraction in layer of the network but distinctly different as a CNN does not have to extract features manually (Li et al., 2022).

CNN’s usually consist of three components that transform input volume into an output volume, this is seen as convolutional layers, pooling layers, and connected layers. These layers are then stacked to create the CNN (Batmaz et al., 2019)

CNN’s are inspired by human visual perception, where a biological neuron corresponds to an artificial neuron in the neural network, this is similar to many other networks and AI in general, human beings and our own biology are a basis for many of the advancements and systems in artificial intelligence. CNN kernels are each receptors that respond to features extracted in layers of the network. These receptors simulate the transmission of electric signals to another group of neurons exceeding a certain threshold (Li et al., 2022).

There are many different types of convolution that accomplish different tasks and approach convolution in a slightly different way, these are Deformable Convolution, Group Convolution, Steerable Convolution and Graph Convolution.

Deformable Convolution is designed to handle irregularly shaped objects. Group Convolution is used to train models with lower GPU performance. Steerable Convolution is intended to handle equivariance in image recognition, allowing linear recognition to adapt to transformations/translations in an image and Graph Convolution allows for graphical data to be processed by a CNN but it has an issue, bottle necking because CNN’s cannot learn data in non-Euclidean space (Li et al., 2022).

A diagram of a neural network

AI-generated content may be incorrect.

Fig 2.2.1 Shows the nodes in an AI model including Convolutional layers. (Li et al., 2022).

Machine Learning, Supervised vs Unsupervised

Machine learning is defined by Arthur Samuel as “The field of study that gives computers the ability to learn without being explicitly programmed.” Machine learning is used to teach machines to handle data efficiently and effectively (Mahesh, 2018). Machine learning itself is an umbrella term of study for numerous different types of machine learning, like Supervised and Unsupervised learning, multi-task learning and neural networks among others as seen in the figure below.

A diagram of a machine learning

AI-generated content may be incorrect.

Fig 2.2.2 Machine learning umbrella (Mahesh, 2018).

In this section we will mainly be discussing supervised and unsupervised learning as opposed to the other types of machine learning, neural networks was discussed above in the Convolutional Neural Networks section and deep learning, another type of machine learning not shown in the figure will be discussed in depth in a later section.

Supervised Learning refers to the machine learning task of a model or machine, learning any type of function that maps a given input to an output based on example input output pairs. It utilises training and testing data, split up by the developer in control of the model and sometimes validation data. Supervised learning algorithms are those which need external assistance from said developer to function properly (Mahesh, 2018). Supervised learning is applicable for prediction or classification and is utilised in many recommender systems as standard.

Unsupervised learning, as opposed to supervised learning, is a system in which the machine is left to its own devices. There are no correct or incorrect answers, the machine and its algorithms are left to discover and present its own structure in the data. Unsupervised learning learns few features from the data itself, and is mostly used for clustering and feature reduction within a system (Mahesh, 2018).

Deep Learning

Deep learning is a field of machine learning based on learning layers of representations, usually using neural networks. Through layer hierarchy, higher level concepts are separated or defined from lower level concepts (Deng & Yu, 2014).

A diagram of machine learning

Description automatically generated

Fig 2.2.3 Artificial Intelligence vs Machine Learning vs Deep Learning (Middleton, 2021).

Deep learning currently produces state-of-the-art solutions to many problems in computer vision, natural language processing and speech and audio recognition (Deng & Yu, 2014).

There are different types of deep learning models similar to CNN’s or statistical models for AI, which will be discussed later in detail. Some of these deep learning models are:

Restricted Boltzmann Machines: A machine which contains two layers of units, these are visible units and hidden units (Salakhutdinov & Hinton, 2009). Visible units correspond to components of observation and hidden units are representative of dependencies between the components of the observations (Ciresan et al., 2010).

Deep Belief Networks: A DBN is a multi-layer learning architecture using a stack of RBM’s (above) to extract hierarchical representation of training data supplied to the model (Hinton, 2009). The DBN uses the stack of RBM’s visual and hidden units to obtain a model to extract features from the input. This is usually unsupervised so it can be beneficial to add in a supervised learning network at the end to achieve classification or regression predictions (Batmaz et al., 2018).

Autoencoders: Autoencoders similar to CNN’s are a feedforward neural network trained to encode inputs into some representation (Hinton & Salakhutdinov, 2006).

They do this using three layers, an input layer, hidden layer and output layer, the number of nodes in the input layer are equal to that of the output layer, because the hidden layer is hidden and unknown to us the number of nodes are not known. The autoencoder reconstructs the input layer as best it can in the output layer using representation gleaned from the hidden layer. During the learning process the network uses two mappings, the encoder and the decoder. The data is then encoded to the input layer and decoded to the output layer giving a prediction (Batmaz et al., 2018).

To avoid useless data being given from the autoencoder which can be common, a denoising factor is used in the original data (Batmaz et al,. 2018).

Convolutional Neural Networks: CNN’s are another type of deep learning model, these were discussed at length in a previous section.

There are other models alongside the four mentioned here that are not mentioned here as well, and all have different methods of doing similar tasks to differing effectiveness.

## Types of Recommender Systems

Collaborative Filtering

Collaborative filtering is a type of approach when building recommender systems, it is the most common approach when building these systems ahead of content based and hybrid models of recommender system. The assumption is that people who agree on their tastes in the past, would agree in the future with newer preferences as well. (Sarwar et al., 2001). In collaborative based systems, the preferences of like-minded neighbour users form the basis of produced recommendations as opposed to individual features on given items or content (Batmaz et al., 2018).

The primary actor of a CF system is the active user themselves (*a*) who looks for a prediction of items in a ranking or rating, it utilizes past preferences as the main indicator for determining correlation or similarity among users, making those users said like-minded users. A CF would then yield referrals to user *a* given the tastes of those similar/like-minded users (Batmaz et al., 2018).

Usually a CF constructs a list of users and items, the system then utilizes these lists and inserts them into a matrix that contains the user ratings or rankings for these items. The CF algorithm then predicts a rating for the item in question or creates a list of *N* items ordered by assumed affinity for user *a* (Batmaz et al,. 2018). These items are then displayed to the user in the way in which the current application or process deems appropriate.

A diagram of a graph

AI-generated content may be incorrect.

Fig 2.3.1 Overview of a CF system (Batmaz et al., 2018).

Content Based Filtering

Content based recommender systems work off the basis of descriptive attributes of items. The system takes into account the previous preferences or likes of the active user to predict and recommend items or content to that user (Van Meteren & Van Someren, 2000).

In content based filtering the main purpose is to recommend similar items or content to what the user has shown affinity for previously. If a user likes an item with keywords such as “oven”, “cooker”, “toaster” or likes a user account on a dating application with pass times like “sports”, “movies”, “alternative music” the content based system would recommend that user with similar affinities or keywords to previously liked items or users (Batmaz et al., 2018).

Content based systems are very efficient when recommending new items into a system. Although there are no ratings or rankings for the new items, if said item contains a description with keywords the algorithm can utilize this descriptive content to recommend this item to users who have liked previously similarly descripted items in the past (Batmaz at al., 2018).

Because content based filtering systems do not use community knowledge from like-minded users they struggle to recommend nuanced and personalized content to the user as there is not enough content in limited groups of keywords as averse to a diverse range of preferences from upwards of hundreds of users (Lops et al., 2011).

A diagram of a person's purchase

AI-generated content may be incorrect.

Fig 2.3.2 Content Based Filtering System (Zainurrohman, 2021).

Hybrid Filtering Systems

Hybrid based systems utilize both collaborative filtering and content based filtering systems together, this is because these systems both have specific and idiosyncratic strengths and weaknesses. Hybrid systems combine these strengths and weaknesses to attempt to create a system which bypasses both systems’ specific problems by exploiting the benefits of the other (Batmaz et al., 2018).

A typical hybrid based filtration scenario would be that of employing content based descriptive information of a new item without any user rating in a CF recommender system (Tran & Cohen, 2000).

Numerous techniques of hybridization of recommender systems have been proposed which can be described as follows (Burke, 2002):

* *“Weighted*: A single recommendation output is produced by combining scores of different recommendation approaches.”
* *“Switching*: Recommendation outputs are selectively produced by either algorithm depending on the current situation.”
* *“Mixed*: Recommendation outputs of both approaches are shown at the same time.”
* *“Cascade*: Recommendation outputs produced by an approach are refined by the other approach.”
* *“Feature combination*: Features from both approaches are combined and utilized in a single algorithm.”
* *“Feature augmentation*: Recommendation output of an approach is utilized as the input of the other approach.”

## Statistical Models in Recommender Systems

Classification

In classification data is attempted to be sorted into classes and predicted to be in one of said classes by the model itself. Used is a training sample of *n* observations on a class *Y* that takes the values 1,2 ….., *k,* where k is the final number of classes present. It then takes predictor *p* variables *X1* to *Xp.* The goal is to find a model for predicting the values of *Y* from the new *X* values. The solution is a partition of the *X* space into *k* sets or classes to find the predicted value of *Y* is item *j* if *X* belongs to a given class *k* (Loh, 2011). Then used will be a model, for example K-nearest neighbours, to determine the scope of prediction, which will be talked about later.

A diagram of a graph and a diagram of a graph

AI-generated content may be incorrect.

Fig 2.4.1 Diagram showing an example of a classification decision tree (CART), featuring classes labelled 1,2,3,4. On the left is partitions and right is the decision tree structure itself (Loh, 2011).

Regression

Regression on the other hand are models widely used across a range of scientific applications and provide a general and malleable approach when describing the dependence of response variables on those of explanatory variables (Fitzmaurice, 2016).

Regression models are exemplary at things classification models struggle with, one example would be explanation. That is, the regression model is used to predict the effect of an explanatory variable on the response, while at the same time controlling for the effects of any other variables that are included inside of the model. This can be used in the example of episodic memory in Alzheimer’s patients. The regression model yields an estimate of the effect of the explanatory variable of interest (episodic memory), controlling for any other factors within the model (age, years of education), hence enhancing our understanding of the relationship between the response and explanatory variable (Fitzmaurice, 2016) .

A green and blue line graph

Description automatically generated

Fig 2.4.2 Simple example of a regression model (Regression Model: Definition, Types and Examples - Voxco, 2021).

Models

In this section discussed will be some statistical models used within AI models inside of recommender systems, these will be, Linear Regression, K-nearest-neighbours for classification & regression and decision trees/random forests for both classification and regression also.

Linear Regression:

Linear regression is a simple and useful tool for predicting a quantitative response. Though it may seem simple or even dull in comparison to newer more modern statistical models it is still a useful tool and widely used statistical learning method (James et al., 2023).

Linear regression is a straightforward approach for a quantitative response *Y* on the basis of a single predictor variable *X.* It assumes there is approximately a linear relationship between X and Y, mathematically written as (James et al., 2023):

**

Fig 2.4.3 Mathematical depiction of a linear regression model (James et al., 2023).

A graph with a line and a red arrow

AI-generated content may be incorrect.

Fig 2.4.4 Diagram depiction of a linear regression statistical model (How Can You Calculate Linear Regression? - Voxco, 2021).

K-Nearest-Neighbours:

The basic idea of a k-NN classifier is to take the nearest neighbours k from a set to determine a prediction or value proposition.

Maybe the most straightforward classifier in machine learning techniques is the Nearest-Neighbours Classifier. The technique is commonly referred to as “the k-nearest-neighbours (k-nn)” classification, where *k* of the nearest neighbours of a given item or variable are used in determining the value or class of that item or variable (Cunningham & Delany, 2021).

A diagram of a graph

AI-generated content may be incorrect.

Fig 2.4.5 Diagram depicting 3 nearest neighbour classifier on a two class problem (x and o) in a two dimensional space (Cunningham & Delany, 2021).

In the figure seen above *q1* will depict that it is class *o* easily as all of its neighbours are in class *o*, for *q2* it will define itself as class *x* as 2/3 of the neighbours are of class *x* (Cunningham & Delany, 2021).

Decision Trees & Random Forests:

From a geometrical point of view, the main principle of tree structured models is simple, it consists in approximating the bayes model by repetitively partitioning the input space into subspaces and then assigning constant prediction values to all objects within each subspace (Louppe, 2014).

A diagram of a computer

Description automatically generated

Fig 2.4.6 A simple diagram of a decision tree, multiples of these up to hundreds and thousands would be used in a random forest (IBM, 2021).

Essentially the model splits data into nodes with possible outcomes from a root node, these are decision or internal nodes and chance or leaf nodes (probabilities of outcomes). The tree follows these nodes to decide upon a prediction (IBM, 2021).

Random forests are in essence a collection of decision trees, making the same decisions, then being collected and averaged in an algorithm to decide upon the best course of prediction.

## Uses of Recommender Systems

Dating Applications

Dating applications come in many different forms, the main two are Tinder and Hinge both owned by the Match Group. These applications use mainly Collaborative Filtering systems with some Hybrid Systems as well to match users together in romantic partnerships (Nader, 2020).

Tinder makes recommendations to their users on their “Top Pick” page and in their general pool on their swiping page. Hinge also has a “Most Compatible” system which matches two users per day together based on a strong recommendation that the two would be romantically compatible (Nader, 2020).

These apps show users one profile at a time giving them the options of either “liking” or “disliking” that given user. If two users are interested and both “like” each other they will be matched together and a chat will be automatically started. These users can then choose whether to interact with one another or not on their own volition (Nader, 2020).

The recommendation algorithms used in these applications are usually hidden and inaccessible to the user and the public, but there is strong reason to believe that those systems are similar to other collaborative and hybrid filtering systems (Nader, 2020).

In E-Commerce

Recommender systems are also used in commerce, these systems are there to recommend items to customers or users so that these users will purchase these products (Schafer et al., 1999).

Companies need to be able to at a minimum, create multiple products that meet the multiple needs of multiple users. This can be done by utilizing things such as recommender systems to help users with choices especially in the realm of e-commerce (Schafer et al., 1999).

Recommender systems are used by e-commerce businesses to suggest products to their customers. Products can be recommended based on a number of factors, demographic of user, analysis of past purchases by said user (Content Based Filtering), top overall sellers on the site or any other analytics designed to narrow down what the user in question may want (Schafer et al., 1999).

Recommender systems can enhance e-commerce in three ways (Schafer et al., 1999):

Browse into buyers:

Visitors to a website sometimes look over the site without actually purchasing any items (skimming), the recommender system can recommend items that are of interest to the user to turn them into a potential buyer.

Cross-sell:

These systems improve cross-sell by suggesting additional products that are related to the product a customer is buying, or that other users have also purchased with said item (Collaborative Filtering).

Loyalty:

Recommender systems improve customer loyalty to a given site by creating a value-added relationship between the site and the customer, by the site learning about the user they can use these recommender systems to increase the enjoyment and efficiency of the user experience on the site by giving them good recommendations, or customising the user experience directly for that specific user. Customers will then return to the sites that deliver the best, most well-tailored experience.

Issues With Recommender Systems

In this section discussed will be the issues with recommender systems today, those discussed will be, ethical issues in dating applications, and the problem of cold-start/sparsity in recommender systems, although there are many other problems with these systems, these will be the ones discussed here.

Ethical Issues in Dating Applications:

Dating apps, as discussed above utilize recommender systems to match users together in a number of ways, using a myriad of filters and variables to decide which users can match with who. One issue with these systems is segregation of users based on ethnic backgrounds or race, and the mental health toll these apps can have on users.

In dating apps, the matching systems in place have been found to segregate users by race, this is due to these peoples exhibiting similar behavioural patterns: users on these platforms seem to segregate themselves based on race and exclude certain races from romantic and sexual consideration, and generally show a preference for white men and women. This can create inequality within the dating culture, and exacerbates already known issues in society by reinforcing them using recommender systems (Nader, 2020).

Another form of segregation is due to appearance, with the split on these apps regularly around 80% men to 20% women, this creates a gross inequality on both sides of this split. Dating apps will exclude users from the pool due to collaborative filtering, if someone is not seen, they wont attain many matches (Nader, 2020). This can cause these users to feel isolated and unwanted as if they aren’t getting likes due to their appearance or personality. This is again a real world problem, only exaggerated by the recommender systems used within these apps.

Solutions to these problems are difficult, because they are caused by human biases and societal issues being exaggerated by the systems we build. Changing our own biases is a start, and building systems that attempt to be more intelligent, and to filter out human bias to create a more equal and equitable system (Nader, 2020).

Cold Start/Sparsity:

A typical recommender system is a responsive system, needing data and historical information about a user to imply predictions on that users preferences. An issue encountered with this is that new users do not have any historical information or preferences data to pull from for these systems, leading to these systems giving poor or completely non-related recommendations to users (Batmaz et al., 2018).

This issue is a difficult and complicated issue to solve, one approach is to transform the high-dimensional and sparse user-item matrix into a lower dimensional and denser set using deep learning techniques (Batmaz et al., 2018).

Conclusions

Recommender systems are incredibly powerful and useful tools for humans to use in modern society, they allow us to be more efficient when doing many tasks, shortcutting any in depth searching we may have to do by bringing us relevant content or information instantly at the press of a button. They have an immense amount of depth, and these systems are only getting more intelligent, more complex, and more efficient.

While these systems are very interesting, powerful and a lot of the time very helpful, we as developers must be conscious of the issues both ethically and technically with these systems, so as they do not give wrong, misleading, dangerous, or ethically questionable recommendations to users who otherwise would not act in those ways, or access that information.

Recommender systems are a tool to be used by humans to help humans, they should help bring us together, or to allow us to skip menial tasks we otherwise would have to do ourselves, they should not be used to reinforce bias, or push unwanted content or information to users.

# Requirements

## Introduction

There are many requirements for FMT, functional and non-functional, working out exactly what those requirements would be was an integral part of the project. Research into each section of the application was done along with what would be needed for each, both functional and non-functional.

Functionally FMT would need to be a full stack application with a working DBMS, back-end, front-end and design system.

Non-functionally FMT would have to work for users themselves, be approachable and easy to understand, while also being secure, reliable and aesthetically pleasing.

## Requirements gathering

### Similar applications

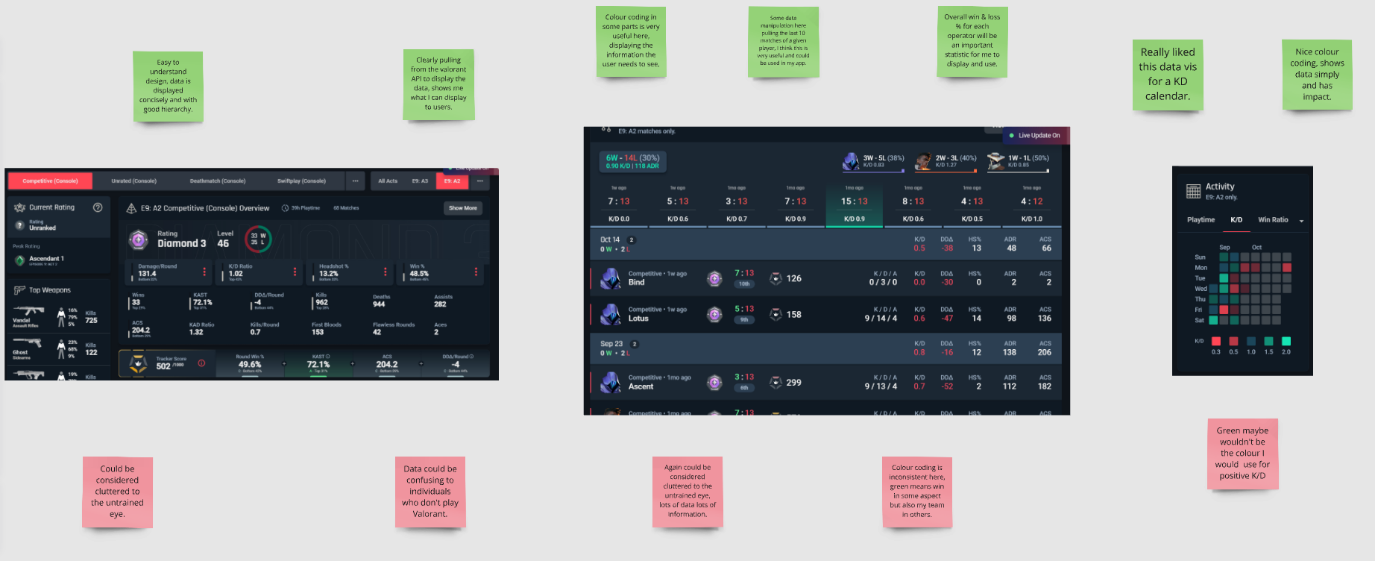


Figure 3.2.1.1 Screenshot of Tracker Network with positive and negative tags for each page.

In the figure above Tracker Network was analysed. Tracker Network is a statistics tracking website that connects to the RIOT API and pulls a players’ statistics into the site, these statistics are then utilised in many ways.

Some of the positives from TRN are their use of graphs and colours to show weather statistics are positive or negative. It is very clear to see how each player stacks up briefly with the way the statistics are displayed.

Negatives are mainly how cluttered the UI can be in parts.

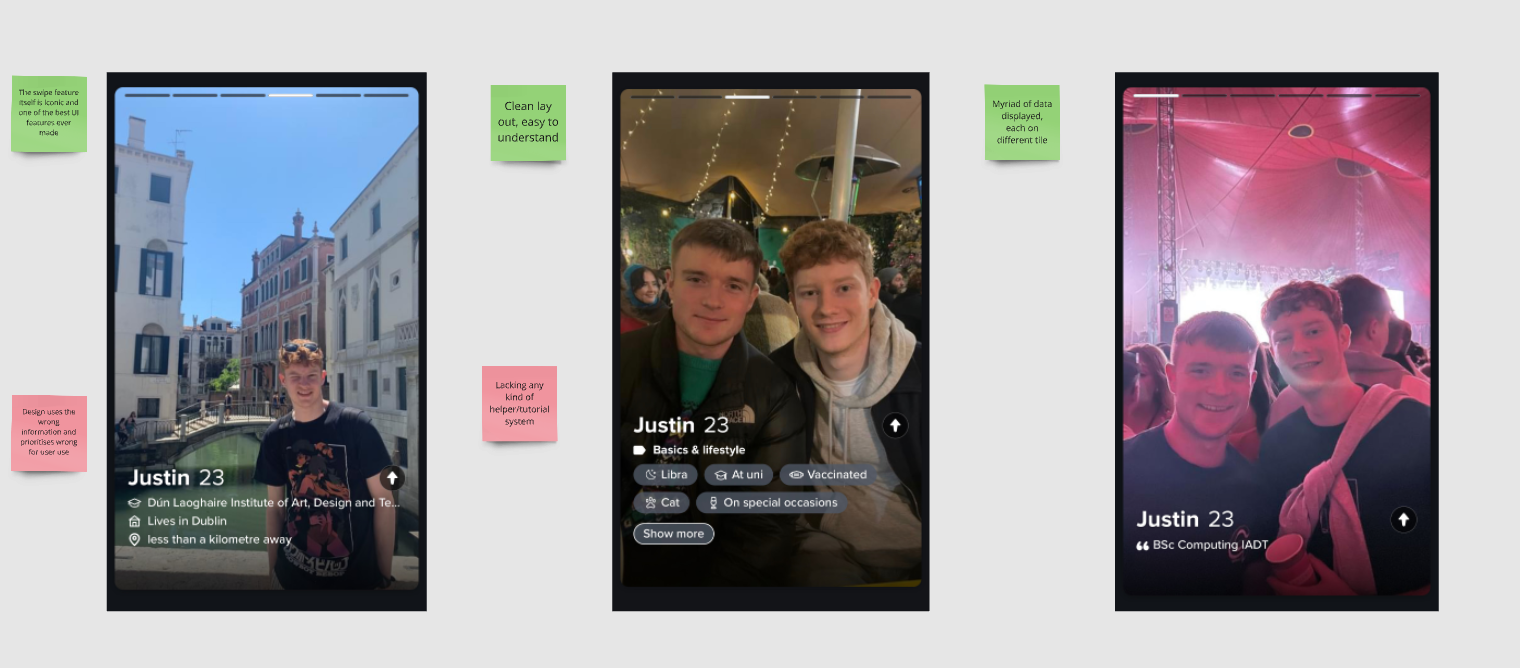


Figure 3.2.1.2 Screenshot of Tinder with positives and negatives.

Tinder is a very popular dating app used by millions of users each day to connect with one another romantically. Tinder will be a large basis for the UI for FMT as they’re both based around matching with users based on data/information displayed and filters for each user to pick from.

Tinder has many positives, as a large and popular app with an enormous amount of capital to back it they have some of the best programmers in the business, thus the front end looks and flows particularly well, being intuitive to users and easy to navigate.

Some negatives I noticed with Tinder were the lack of a helper page having moved through the original tutorial, and the non-existence of a redo feature for likes, meaning once a person is liked there is no way to undo that decision if for instance it was a mistake like.

### Interviews

Interviews were conducted early in the project cycle, three were completed. These interviews are very valuable to understand if the application would be useful to certain individuals and to try and find any blind spots in the project that can be addressed.

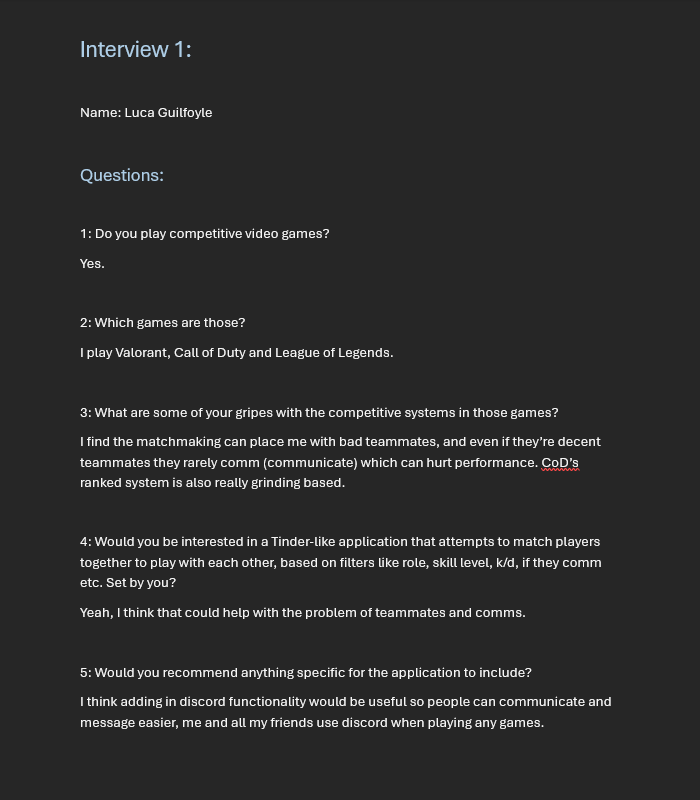


Figure 3.2.2.1 Interview number 1

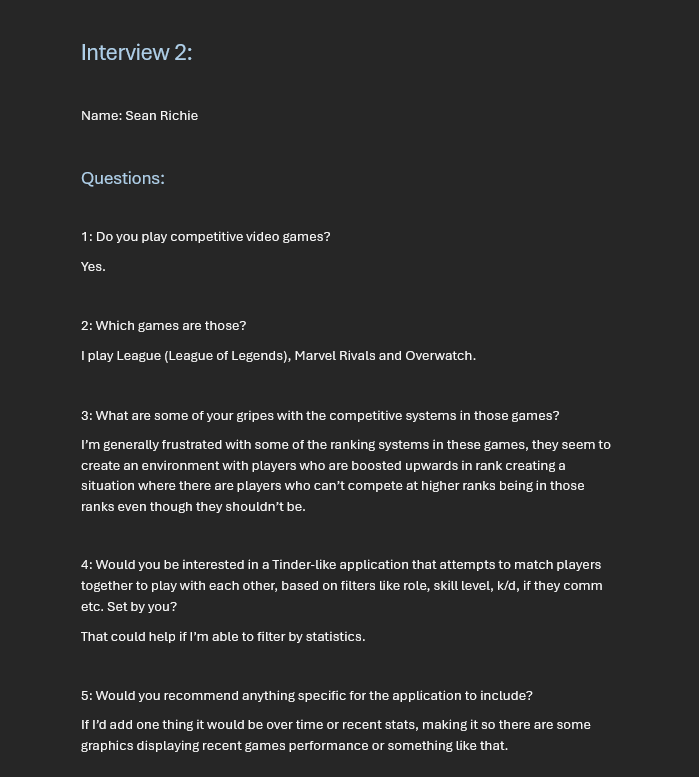


Figure 3.2.2.2 Interview number 2

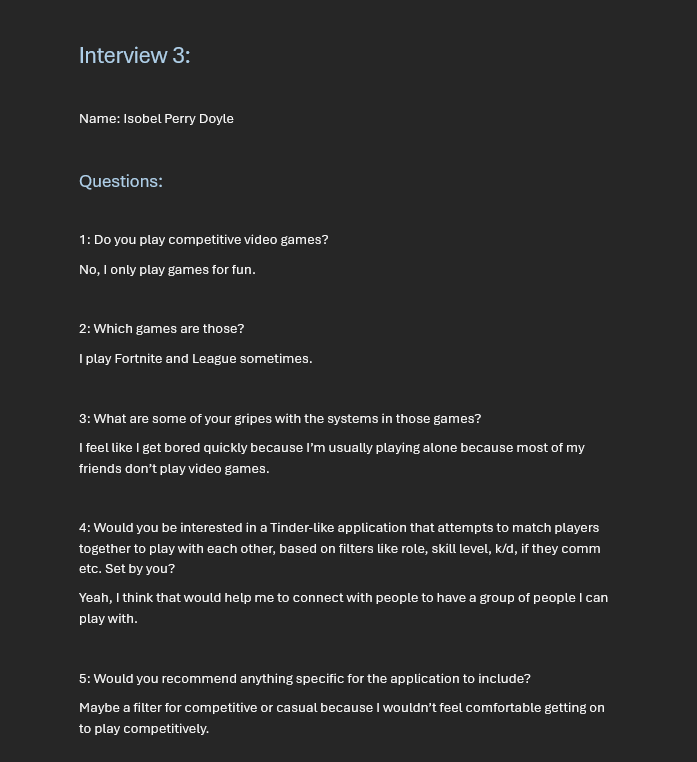


Figure 3.2.2.3 Interview 3

In the interviews it was found that each person was searching to connect with other people, and that this is where these games lacked. Either by underperforming matchmaking or by lack of communication. They think that FMT can provide a space to connect with other players and find community on these games.

Some of the things recommended were data analytics, extra filters and discord functionality which will all be taken into consideration.

## Requirements modelling

### Personas

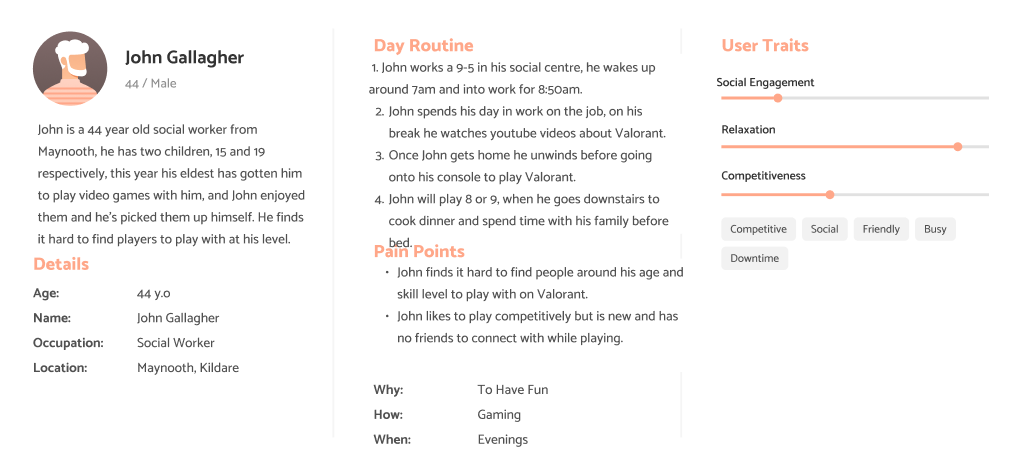


Figure 3.3.1.1 First Persona

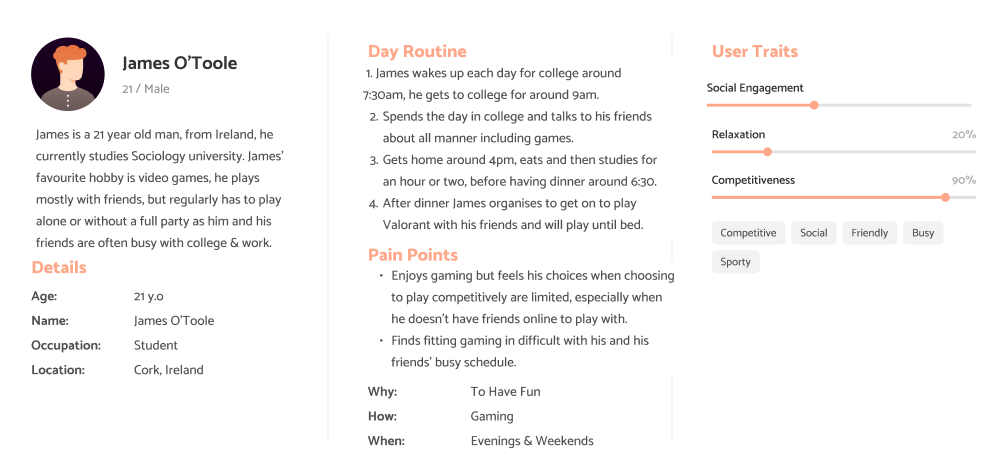


Figure 3.3.1.2 Second Persona

### Functional requirements



Figure 3.3.2.1 Functional Requirements

### Non-functional requirements



Figure 3.3.3.1 Non-Functional Requirements

### Empathy Maps

Empathy maps are an important exercise to understand how users may approach your application and how they may feel while using it. They are designed to help developers and designers understand the emotional journey users may go through while encountering different areas of the application, from possibly apprehension while approaching for the first time, to relief or joy when finding their first match, as will be in FindMyTeam.

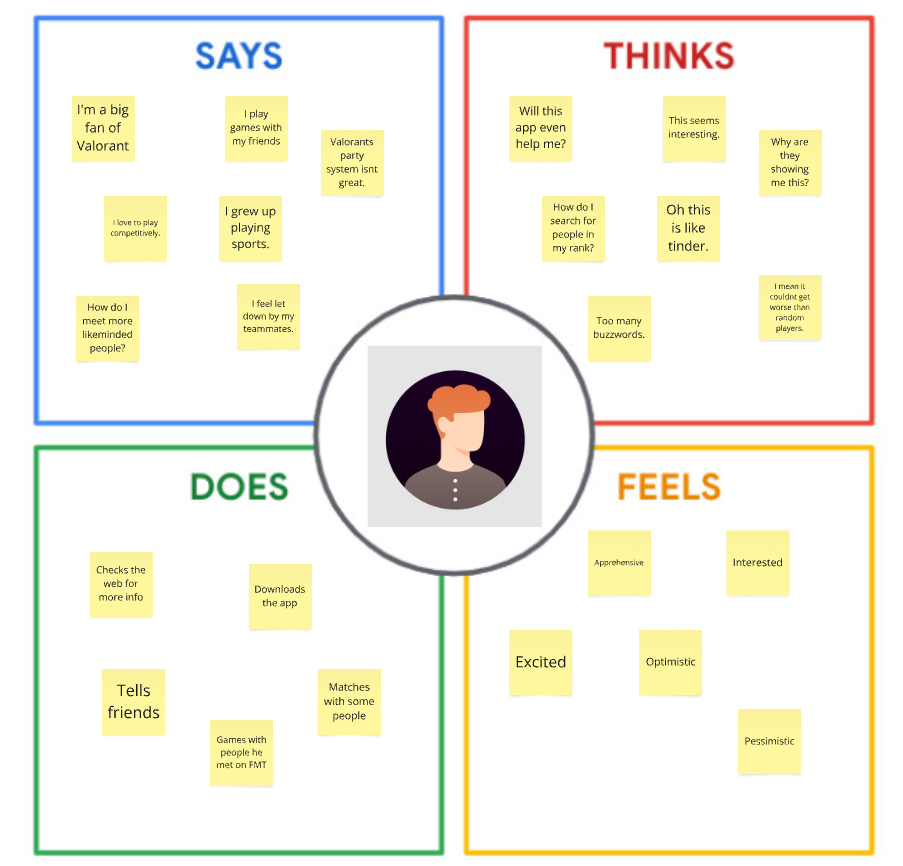


Figure 3.3.4.1 Empathy map for James

James finds approaching FMT very easy, he has experience with web apps since a young age and is an avid gamer, this allows him to pick up FMT’s functionality very quickly.

James is attracted by the simplicity and familiarity of the app, finding it similar to Tinder.

He revels in the idea of playing competitively and tells his friends about it soon after finding out.

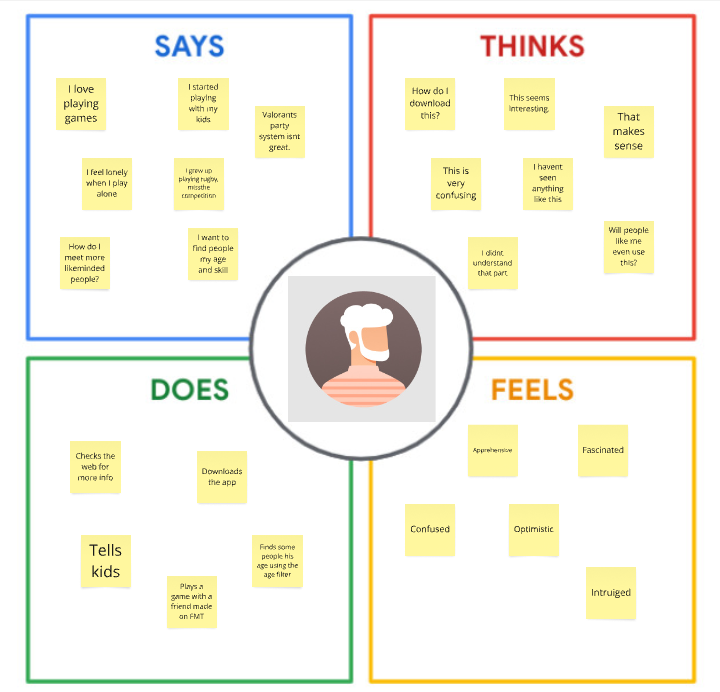


Figure 3.3.4.2 Empathy map for John

Both users are looking to connect with others for slightly different reasons through FMT.

John has a steeper learning curve than James to get to grips with FMT, and he wants to connect in a more casual setting. John is more pessimistic and somewhat confused when approaching FMT, he will need some help via a helper page or tool tips.

While pessimistic originally John warms to FMT’s ideas and tells his children about it afterwards.

### Journey Maps

Journey maps can help us understand users’ needs from the point of view of a user trying to complete a specific task, this will help with the understanding of user flow through the app.

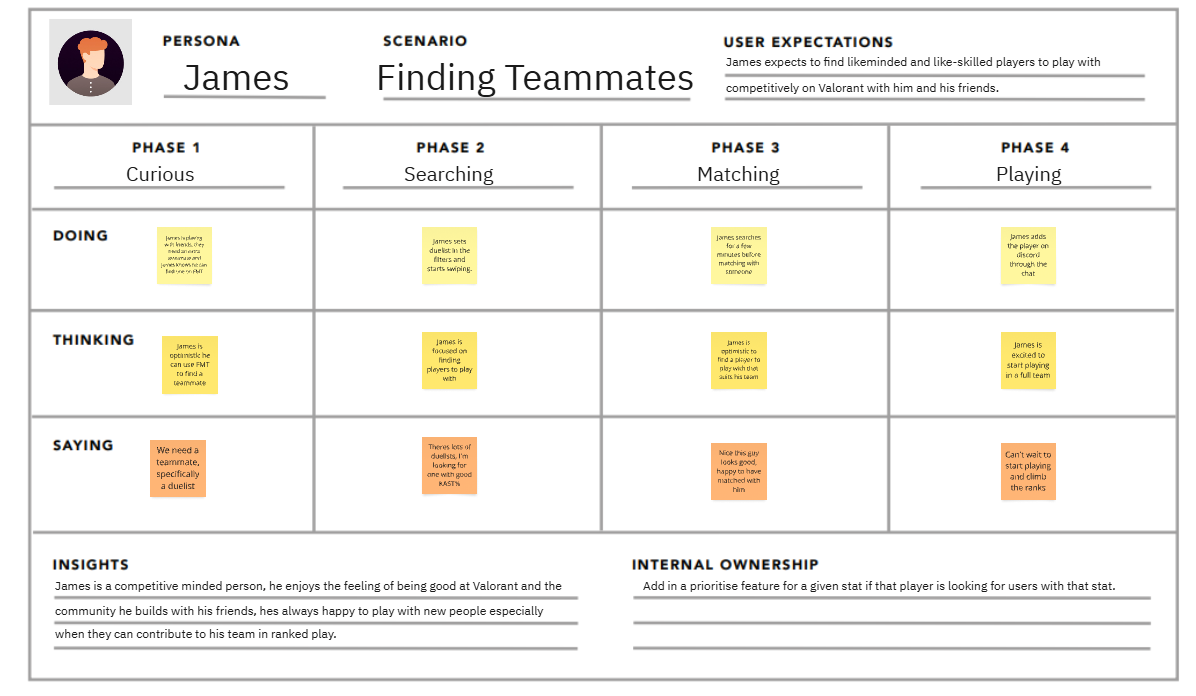


Figure 3.3.5.1 Journey map for James

James is much more interested in micromanaging his preferences in FMT, he wants efficiency when using the app. Adding a sophisticated filtering system will help him, this will enable him to find users who can contribute directly to his squad as soon as they start playing together.

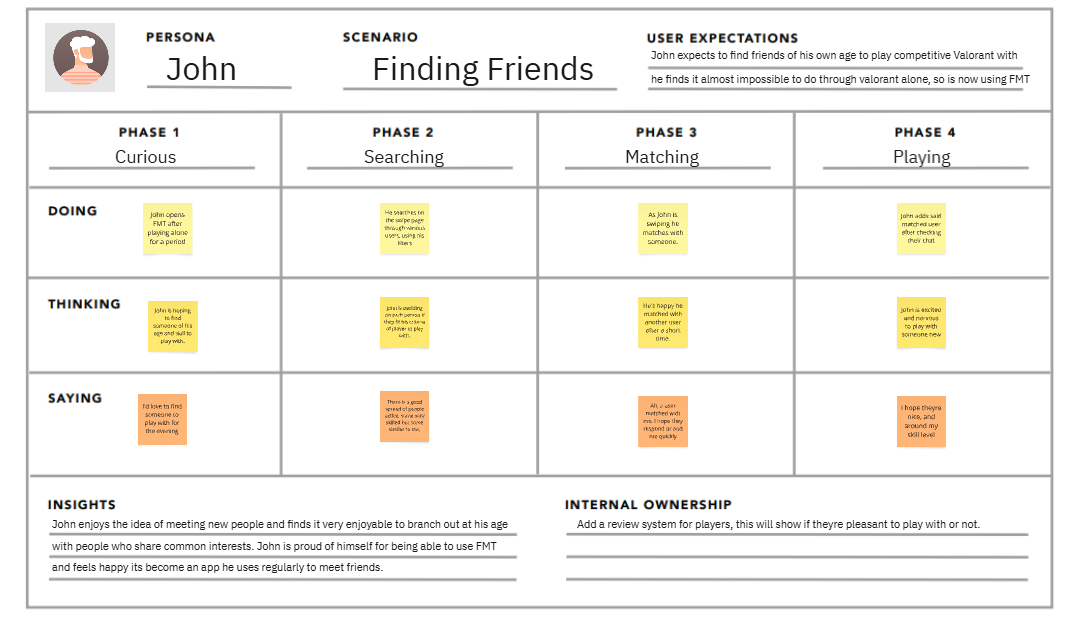


Figure 3.3.5.2 Journey map for John

John as opposed to James enjoys browsing on FMT more, he doesn’t mind taking his time with the FMT experience and enjoys looking at different peoples statistics while using the app, this allows him to find people he really wants to play with and branch out, finding people in or around his skill level.

### User Scenarios

Scenarios allow for finding gaps in the application through specific task analysis through the eyes of each persona user, having a question and answer-based dialogue at each checkpoint allows for a conversational and procedural look through FMT.

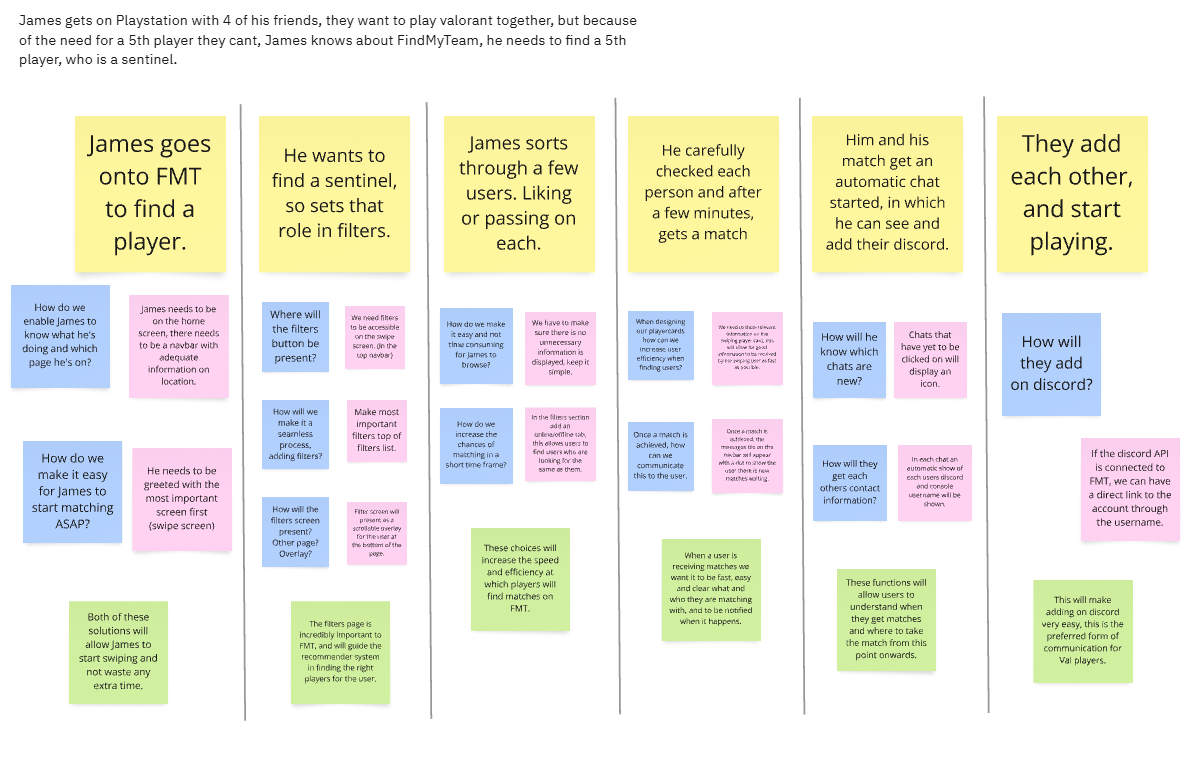


Figure 3.3.6.1 Scenario for James

James wants to hop straight in and find a specific player to play with, efficiency is his main need, he wants precise and simple to read data so he can use his learned skills to find a player.

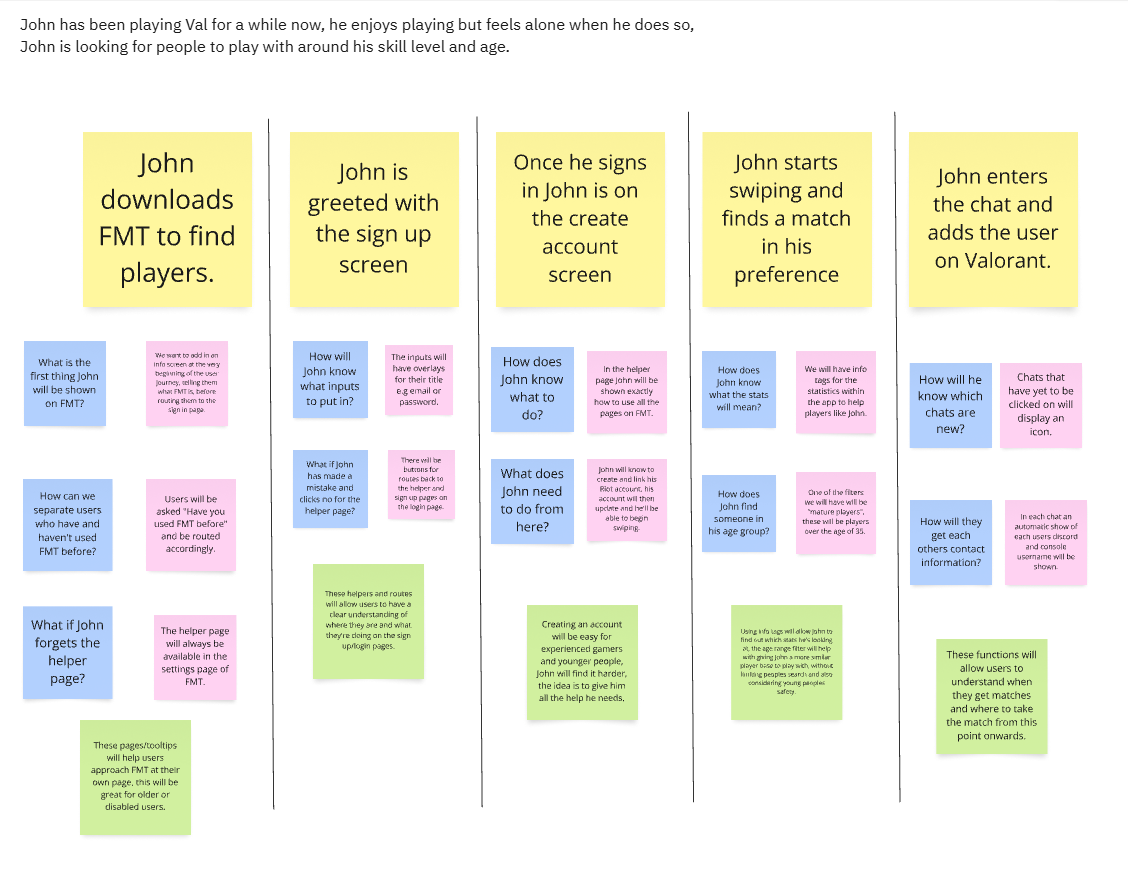


Figure 3.3.6.2 Scenario for John

John on the other hand is more interested in learning and experiencing FMT as an app, he doesn’t know exactly what he’s looking for and needs help to understand FMT, giving John a way in which to find the help screen from his homepage will be beneficial to help him remember how FMT works when he re-visits the site.

## Feasibility

FindMyTeam will use several technologies both in its build up and implementation. As of now none of these have conflicted with any others.

For the back end PyFlask is being used along with many Flask modules and toolkits like SQLAlchemy, Flask Query and others. For the DBMS SQLite is being utilised as it is a lightweight file-based DBMS not requiring a server to use while also being relational. Moving into deployment a move to PostgreSQL will be facilitated due to limitations regarding file based databases and hosting.

The choice for front end is ReactJS with Vite as it is widely used and extremely popular amongst developers. NPM, react router, Vite and others are being used in conjunction with React in the front-end. Shadcn tailwind is the preference of UI framework being used.

In addition, the technologies of GitHub, Insomnia Client, Miro, VSCode and Figma have been utilised in the development of FMT.

## Conclusion

Requirements is an integral part of the development of any project, knowing what is needed both by the developers and users to create and deploy said application. In this section many things were achieved, an understanding of the technologies to be used in FMT, how they would work in conjunction with each other and possible feasibility issues were thought out.

Also dissected was the user experience, through interviews with real people and a survey, along with an exploration through the user experience using personas, scenarios, journey maps and empathy maps, which helped to understand how each user would feel and think moving through the application.

All this together helped immensely to discover gaps within FMT where there were previously blind spots to users and developer’s needs.

# Design

## Introduction

FindMyTeam is an application designed to match users together similarly to a dating app (e.g. Tinder). These users will each be playing the same video game (League of Legends) and FMT will use their statistics and role as a basis for filtering through users to find those who match the preferences of searching users. Users can then match with each other, add each other to groups where their discord information and a link to their account will be shown so they may go, team up and play LoL.

## Program Design

### Technologies

Technologies being used to create FindMyTeam:

* Python Flask – For back end development, to manage logic and errors seamlessly.
* React JavaScript – Front end, to build a user friendly and efficient website.
* SQLite (development DB) – Relational database management.
* PostgreSQL (deployment DB) - Relational database management.
* ShadCN & Tailwind – Design framework to build a utility first, accessible application supported by the ShadCN library of components.
* API ingestion (RIOT API, Discord API) – For user connectivity and statistics ingestion.

Other technologies that were considered for the creation of FMT were:

* **Neo4j:** Considered because of the use of data analytics tools in conjunction with statistics drawn from the RIOT API but deemed unnecessary due to its relational complexity vs standard SQL DBMS’. More suitable for larger scale applications requiring deep and complex relationships between tables.
* **React Native:** Considered due to the relevancy of having applications be accessible on mobile devices but ultimately set aside since LoL is a PC only video game and would require users to be moving from device to device, instead of staying on the same platform. More suited to console exclusive titles where the ability to access discord links or download apps on the console itself is non-existent.

FindMyTeam required a full stack approach to development, this is shown in the technologies used ranging from DBMS technologies, through back end with PyFlask, front end with ReactJS-Vite and finally to ShadCN and Tailwind for design. Two third party API’s will be ingested, the RIOT API provides user statistics for be displayed in FMT and the Discord API gathers the user id and applies it to FMT’s user model column titled “discord\_id” allowing for a direct connection to be made between the two and users a link to another users’ discord.

### Structure of Python Flask & React JavaScript (2 pages)

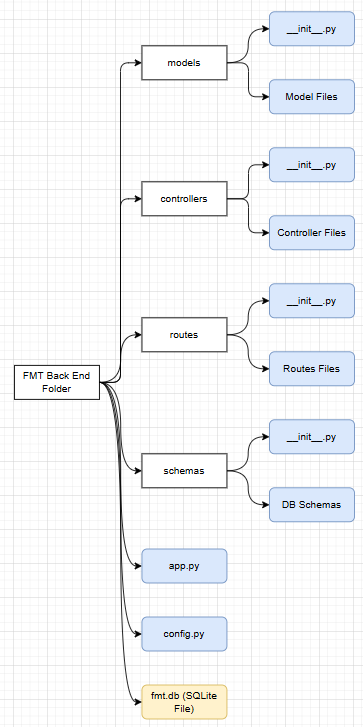


Figure 4.2.2.1 Flask file structure during development

In the figure above seen is the folder structure for the flask back-end with the SQLite DB file included as it sits within the back-end folder. The MVC model is being used as seen with the Models, Routes, and Controllers, the Views section will be handled by the React front end as seen below.

Inside each of the folders is an \_\_init\_\_.py file, which intakes all the folders other files (seen as ‘Models Files’ or ‘Controllers Files’ in the diagram but will be titled as ‘User.py’ or ‘match\_controller.py’ etc. in the actual project) and exports these files to be used by other folders and the app.py file.

Also seen are the schemas for the SQLite database along with its database file titled ‘fmt.db’. Once deployed this file would cease to be used and a linkage to a hosted PostgreSQL database would be implemented within the config.py file.

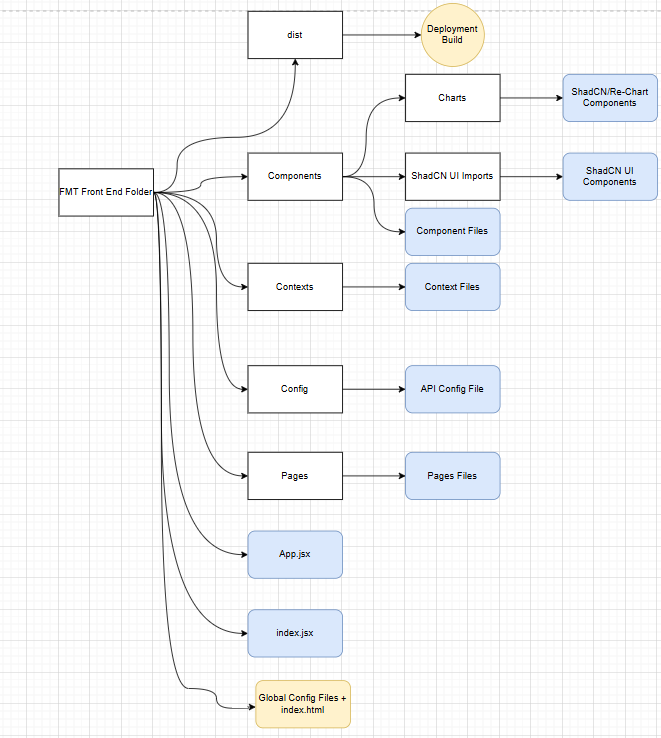


Figure 4.2.2.2 Front end file structure for FMT.

Above seen is the file structure for the front end of FMT. The front end consists of pages which import components from the components folders to display information to the users. These pages are then displayed through the router in the App.jsx file seen in the base folder for the application. These are all then fed to the index.jsx component which wraps this single <App/> component with an authentication provider and the navigation sidebar. Which is all then rendered through the index.html file. This is all compiled and built in the dist folder at the top to be displayed by Vercel and hosted on their servers.

### Design Patterns

FMT uses the MVC model of design, Flask handles the Models and Controllers, while also having routes separate from the controller. This is unusual for Flask as usually the routes themselves act as controllers but for this project they have been split into separate entities with the routes and controllers having their own folders, this is due to simplicity and prior learning with frameworks where the routes and controllers were split up. 011111

The view section of the MVC design pattern is handled by the react front end, which will control all the user interface and will connect to the back end through HTTP URL routes using axios.

### Application architecture

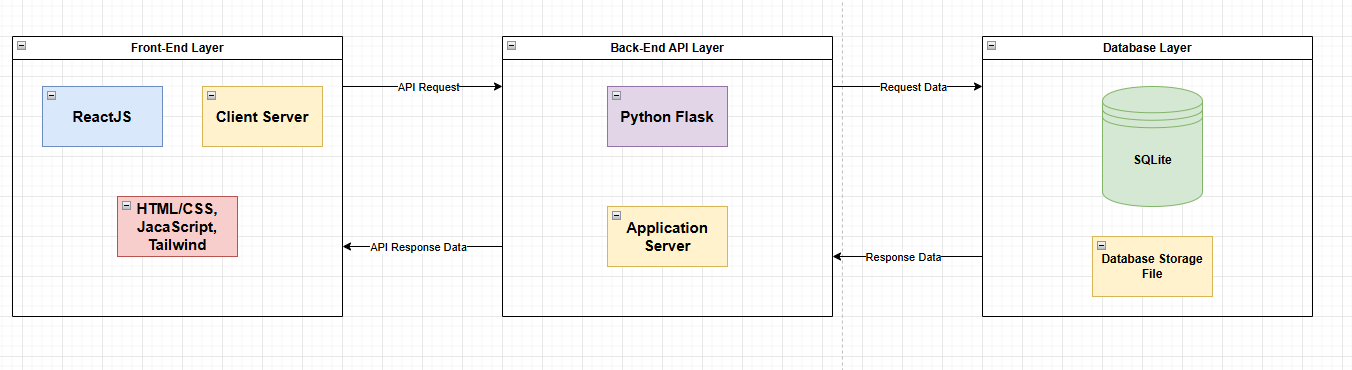


Figure 4.2.4.1 Application architecture diagram for FMT during development

In the development build of FMT the SQLLite database was swapped for a PostgreSQL DB, thus the above image describes the development phase, although the change is minimal and in the diagram would be in name only.

### Database design

ERD’s and Database Schemas are incredibly useful ways of disseminating how the database and back-end structure of an application will work. They show things like table and data structure along with variable types, lengths, names and which ones may or may not be foreign and primary keys. Both are also very useful for understanding the relationship between each table and how the back end will need to be structured for the entire application to function properly.

Below are a Database Schema with all database tables including pivot tables and accompanying relationships, also included is an ERD of the more basic table structure along with the relationships between each of these tables laid out.

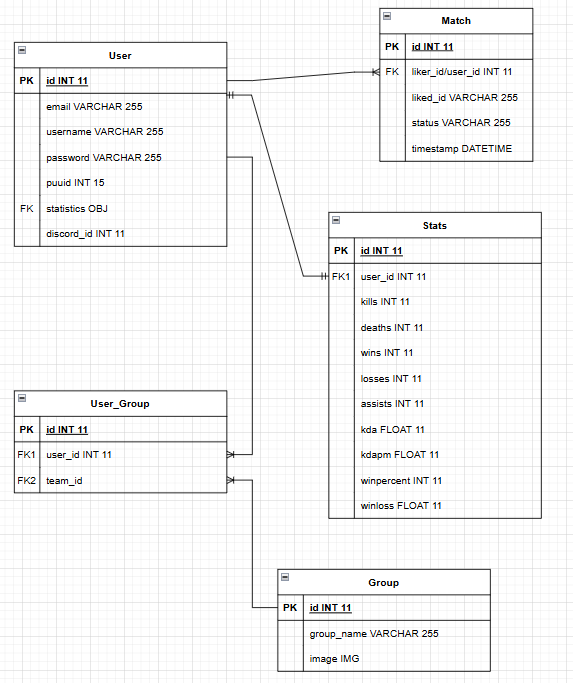


Figure 4.2.5.1 Database schema for FMT

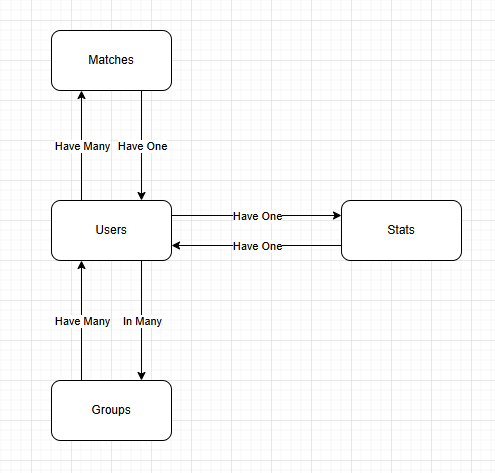


Figure 4.2.5.2 Database ERD for FMT

## User interface design

The UI for FindMyTeam was developed using Figma, app.diagrams.net and sketched models to create an idea of how the front end would look. This is an imperative part of the project to make sure once the design phase is reached it can be worked through smoothly.

### Wireframes

Wireframes are an important way to see how FMT will look moving into the implementation phase of programming, multiple versions of the wireframes were made including higher fidelity prototypes including colour and typography.

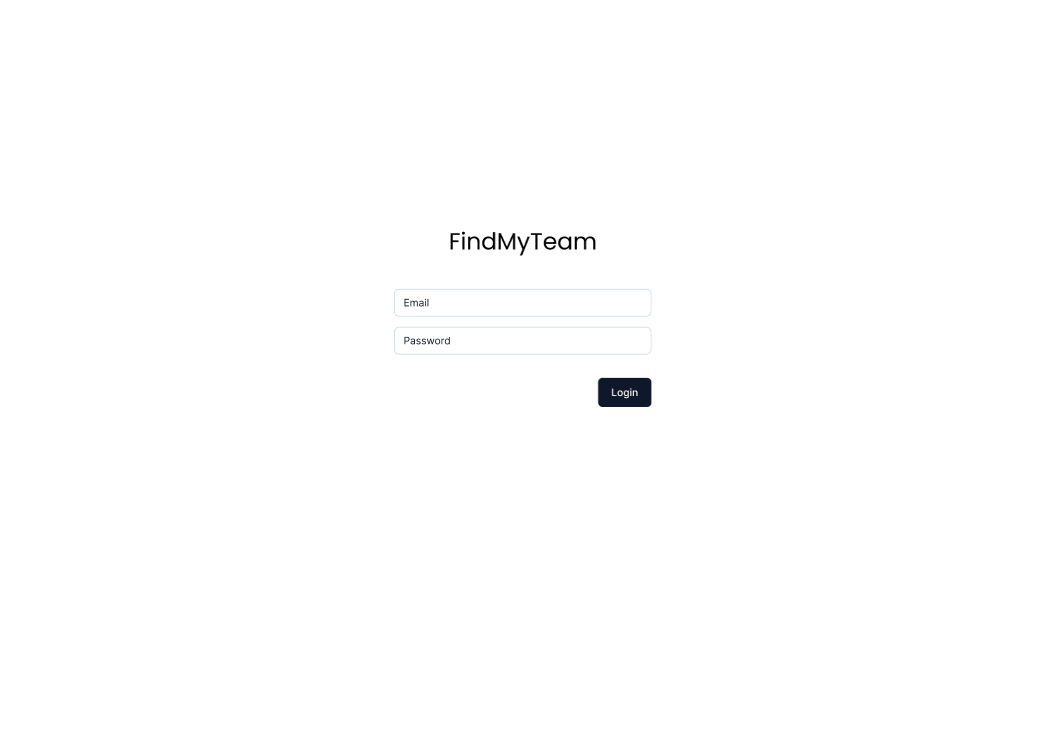


Figure 4.3.1.1 Login wireframe

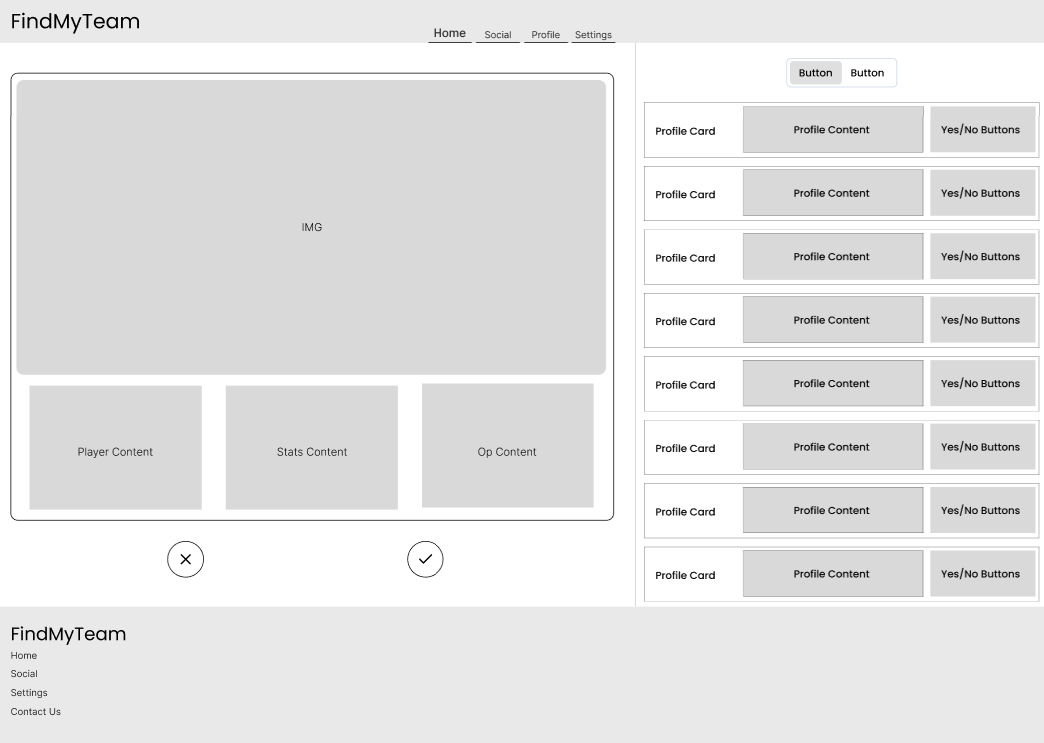


Figure 4.3.1.2 Homepage wireframe

The homepage for FMT has multiple elements seen above. On the left is FMT’s main matching element, similar to that of Tinder’s swipe feature this box will display user information and statistics to the searching user. This will allow them to decide whether to like or pass on said user through the buttons below.

On the right lies the matching box, this will show users who has liked them and will give them the ability to respond with a yes or a no. If the button above the stack is clicked on instead of users who have liked them, a stack of card will show up of users they themselves have liked, and will give them the opportunity to rescind likes if they choose to.

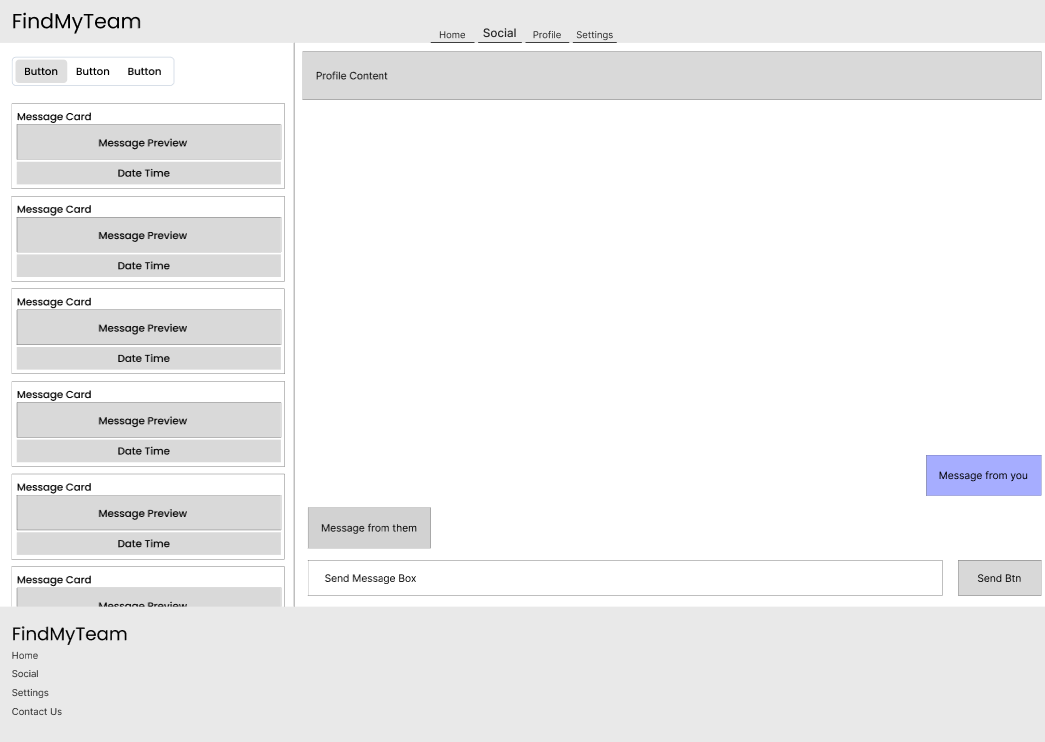


Figure 4.3.1.3 Social page wireframe

On the social page two main elements are present. On the left the cards of matched users will appear, these are users which have both liked each other, each user will be able to click on their matched users’ cards to see their profile page in which their discord and riot tag will be shown.

On the right the “chat” box is displayed, the chat box itself does not have functionality within itself to send and receive chats but instead is automatically generate upon creation of a group, in this chat each user included in the group will have their user discord card and user riot card shown. The discord card will link directly to the users discord and the riot account will display their riot username so as they can be added on League of Legends.

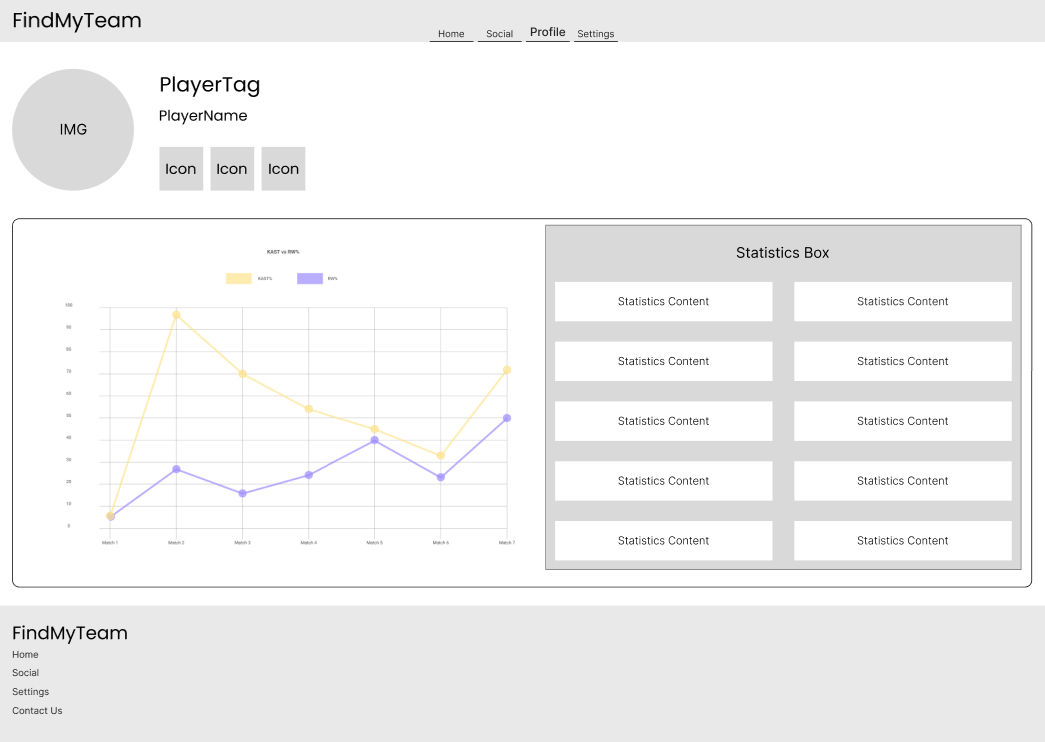


Figure 4.3.1.4 Profile wireframe

The profile page of FMT is an important page and will display information on each user on FMT. On the left is a analytics graph, displaying the users past 5 matches of K/D, this statistic itself may be subject to change in the development of FMT but analytical graphs like this will be an important part of the experience.

On the right displayed is the users statistics, this will be the same statistics displayed on the users’ matching card displayed to other searching users looking to match with them.

Also displayed at the top is the user information, including their discord link, RIOT username, and FMT username.

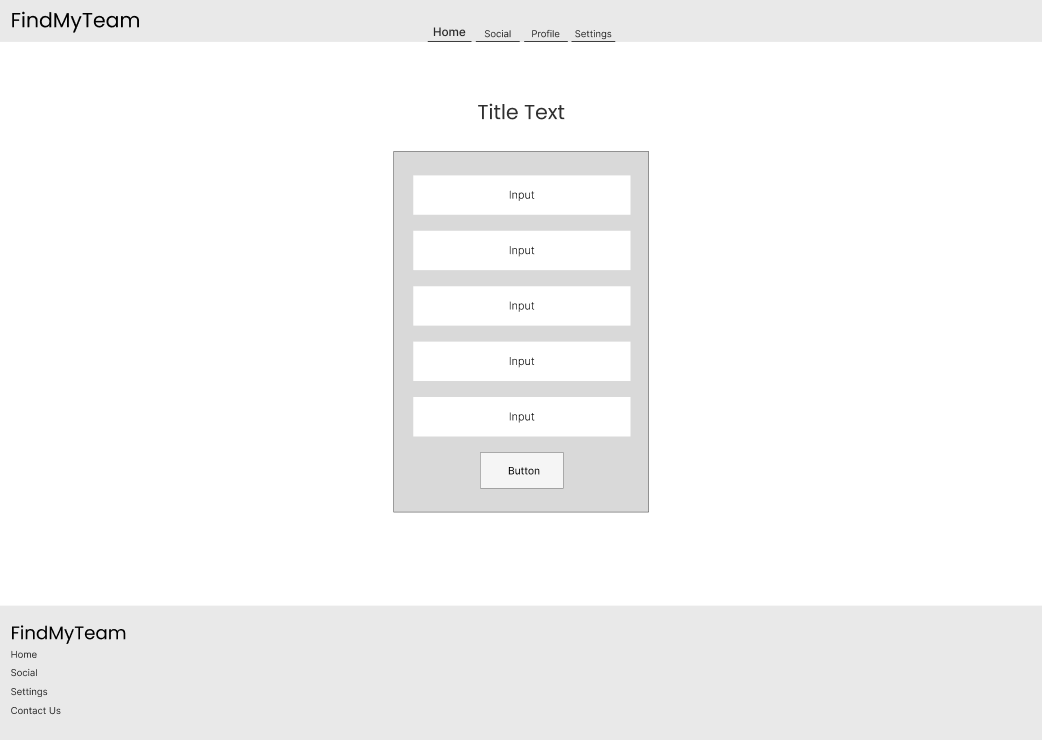


Figure 4.3.1.5 Filters wireframe

Finally the filters wireframe. Shown above is a representation of the filters element which will be incorporated into the homepage to allow users to specify their search preferences.

### User Flow Diagram

The user flow diagram shows how a user will move through the application, they are very helpful for putting the developer in the shoes of the user to find gaps in the design of the app, and account for all use cases.

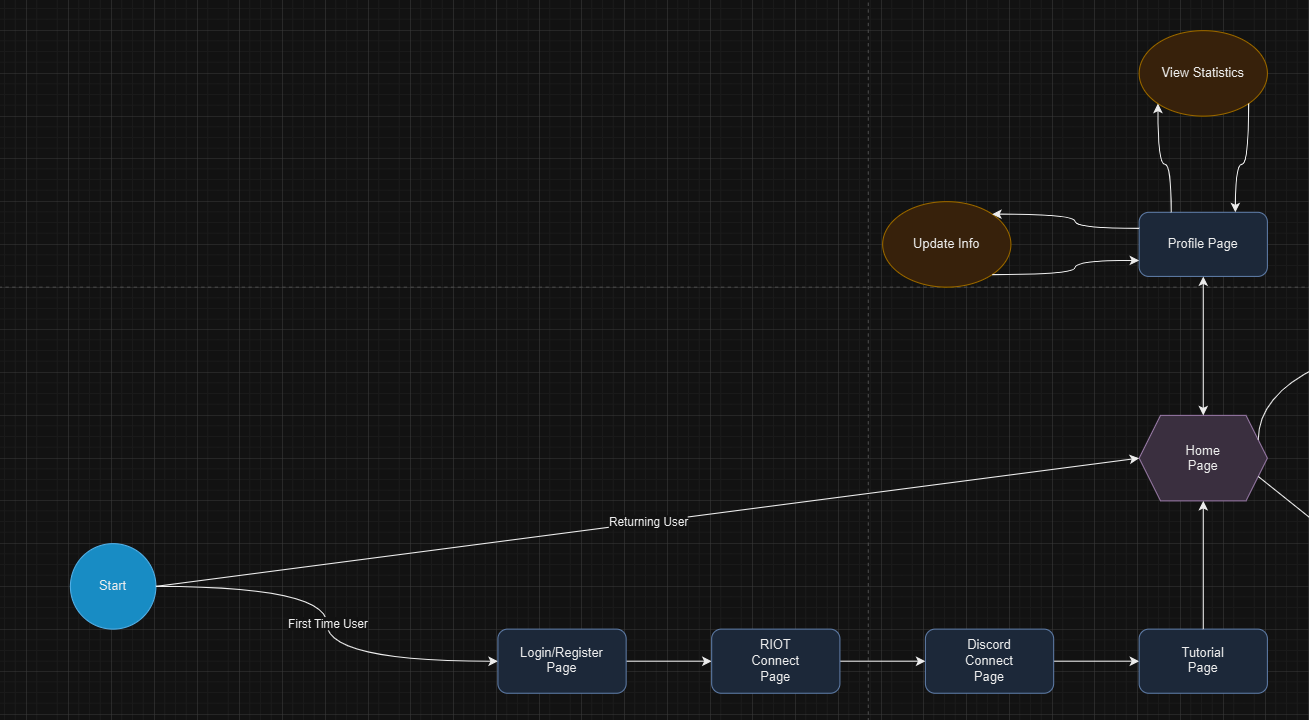


Figure 4.3.3.1 First section of user flow diagram

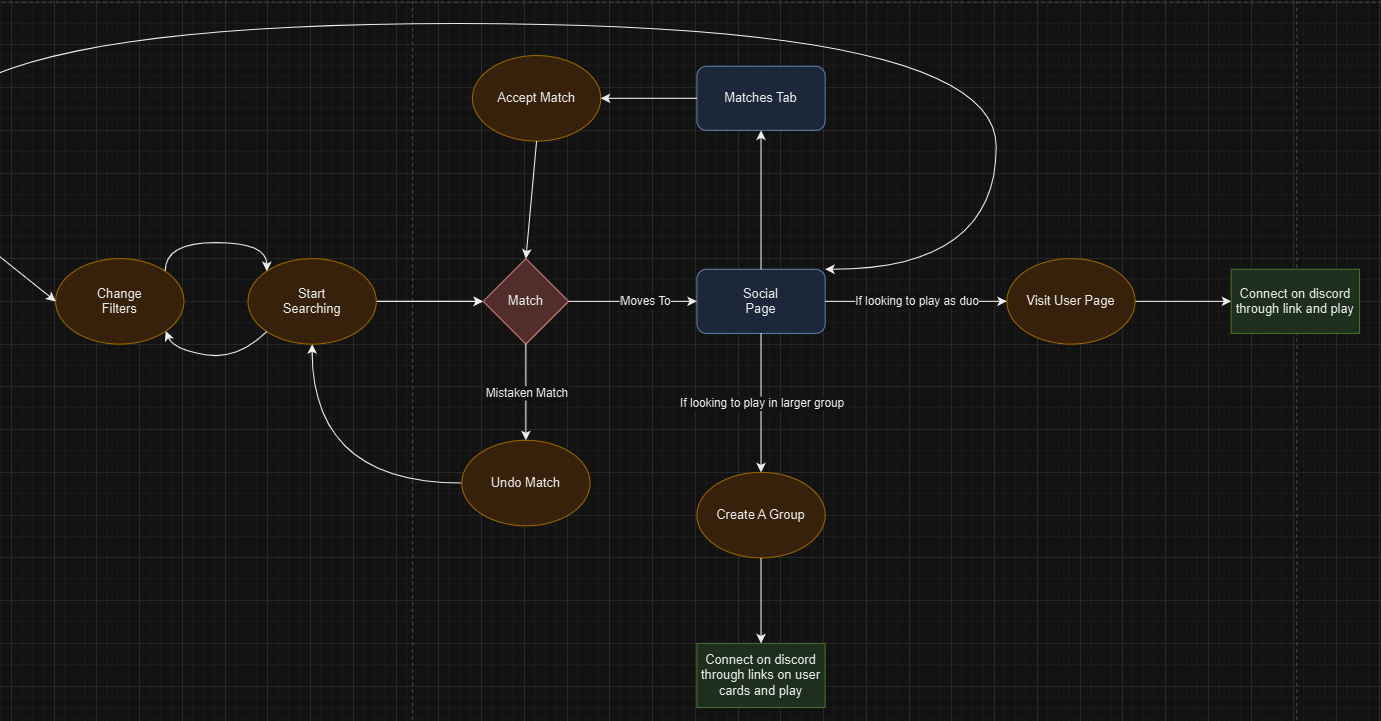


Figure 4.3.3.1 Second section of user flow diagram

### Style guide

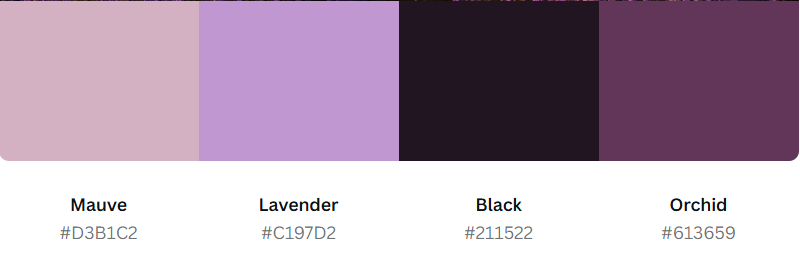


Figure 4.3.4.1 Colour palette Pasture of Dreams for FMT

For the colour palette for FindMyTeam the above was chosen, the mix of subdued and calming colours like lavender and orchid will add a level of calmness to FMT, the subtle mauve will be very useful for differentiating different components and sections without being overwhelming.

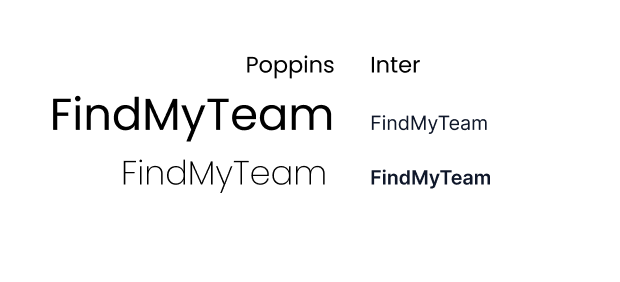


Figure 4.3.4.2 Font choice of Poppins (Headings) & Inter (Body)

For the typography Poppins was chosen as the headings font and Inter was chosen for the body, these two fonts are both sans serif and add a cleanness and professionalism that should be conveyed while using FMT.

## Conclusion

The design section helps developers understand how their application will look and behave, FindMyTeam was helped immeasurably by the work that went into this section, between finding gaps in knowledge, understanding different users and user use cases and the actual look and feel of FMT. The helper/tutorial page was thought up in this section, along with the undo feature for matches, after finding that similar applications did not feature that function for users, showing a clear gap.

# Implementation

## Introduction

FindMyTeam was developed using many different technologies:

**Python Flask –** Back End

**React JavaScript –** Front End

**SQLite –** DBMS

**Tailwind/Shadcn –** UI Design

## Development environment

**VSCode –** Code Compiler/IDE.

**Vite –** Local Development Server/Bundler.

**GitHub –** Version Control.

**Insomnia –** API client & development environment.

VSCode was used as the code compiler/IDE for FMT.

Vite was utilised as the code bundler and local dev environment for the front end code.

GitHub was used for version control, in case of any large scale errors so that a version rollback could be completed & with Vercel to host the code of both front and back end from their repos.

Insomnia was leveraged to develop and test API calls while building the back end of FMT.

# Database Development & Implementation

## SQLLite

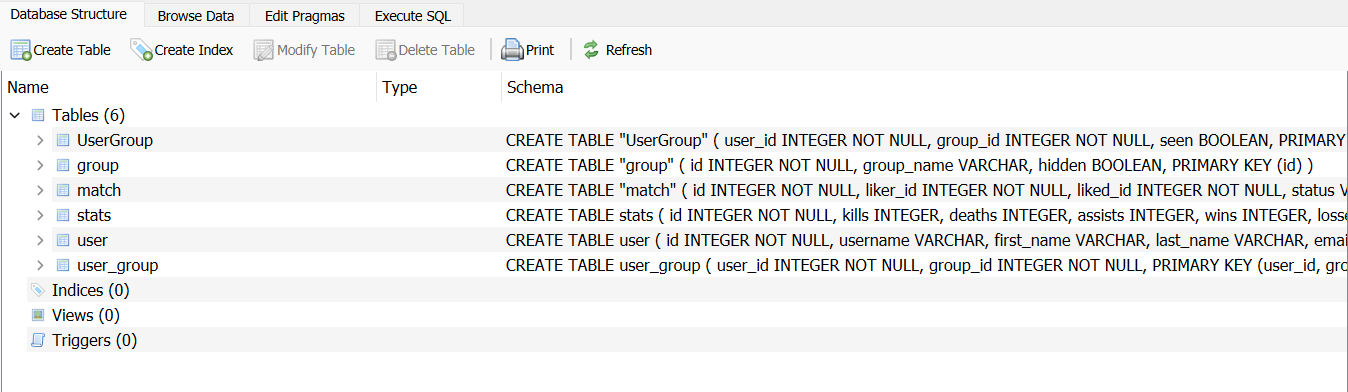


Figure 6.1.1 SQLLite database structure in DB browser

During the development of FindMyTeam SQLLite was selected as the DBMS for the back end written in Python, this was due to SQLLite being highly functional and integrated into Python Flask which was the framework used for the back end.

SQLLite is a file based DBMS, this means that the entire DBMS is stored inside of the backend in a file with a name similar to “database.db” for example. This means that SQLLite is serverless and does not require the running of a server, other DBMS systems like MySQL would require a server to be running in conjunction with the back end to host the DBMS, this could be local or cloud hosted.

The fact SQLLite is file based allows it to run on the locally hosted backend server instead of needing its own, making the use of it during development advantageous.

## PostgreSQL

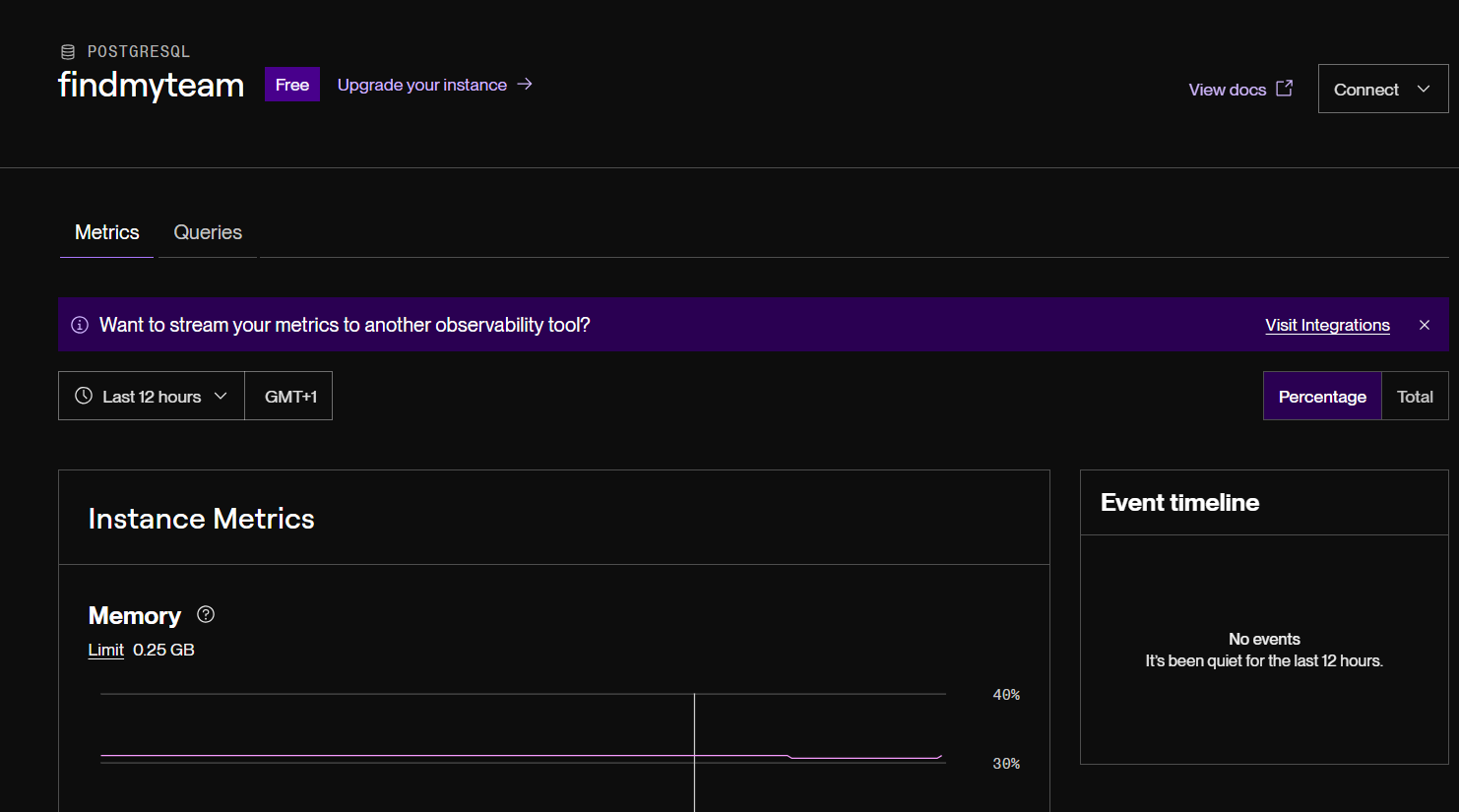


Figure 6.2.1 PostgreSQL database hosted on Render

For the deployment of FindMyTeam it was decided to switch to a PostgreSQL database hosted on Render.com. The reason for this switch was due to deficiencies in SQLLite’s file based data storage, the fact that Render had an in house deployment provider for PostgreSQL databases and Flask and SQLAlchemy having in built functionality to work with PostgreSQL database systems.

Because SQLLite uses a file based system when hosted it will be included with the backend on its hosting service. This can cause issues with read and write queries to the database and can cause data loss at random intervals and during any deployment/re-deployment of the backend.

Due to this the switch to PostgreSQL was decided. Render is a hosting platform for all types of web apps ranging from static sites to full stack dynamic sites like FindMyTeam. Therefor Render was chosen to host the database and backend of FMT.

Creating the PostgreSQL DB was a simple task, simply going onto Renders dashboard and following through the steps of naming and setting URL’s for the DB was all that was required.

Transferring to PostgreSQL in the backend was a seamless move and took very little time, a change was made to the routing in the config (see below) to connect it to the new DBMS in place of the now depreciated SQLLite file which completed the move over.

   SQLALCHEMY\_DATABASE\_URI = 'postgresql://findmyteam\_user:BaKKK9aZzc10NBDtOgmfvPixoBxctPvx@dpg-d079b72li9vc73f3j350-a.frankfurt-postgres.render.com/findmyteam'

# Back End Development & Implementation

## Introduction to Python Flask



Figure 7.1.1 Certificate of completion for the Building RESTful APIs with Flask course on LinkedIn Learning.

For development, PyFlask was chosen to be used as the back end framework to create an API to manage data for the application. Firstly PyFlask would have to be learned as no prior knowledge was held of either Flask or Python in relation to building an API for production level applications. To do this a linked in learning course was undertaken to upskill the development team and build a knowledge foundation for the development of FindMyTeam.

## User

### Model

class User(db.Model):

    \_\_tablename\_\_ = 'user'

    id = Column(Integer, primary\_key=True)

    username = Column(String, unique=True)

    first\_name = Column(String)

    last\_name = Column(String)

    email = Column(String, unique=True)

    password = Column(String)

    bio = Column(String)

    role = Column(String)

    discord\_id = Column(String ,nullable=True)

    riot\_name = Column(String ,nullable=True)

    riot\_tag = Column(String ,nullable=True)

    riot\_puuid = Column(String ,nullable=True)

    riot\_region = Column(String ,nullable=True)

    rank = Column(String ,nullable=True)

    group\_associations = db.relationship('UserGroup', back\_populates='user', lazy='dynamic')

    groups = db.relationship('Group', secondary='UserGroup', viewonly=True)

    stats = db.relationship('Stats', back\_populates='user', uselist=False)

The user model was the first model developed for FindMyTeam in the first few sprints of development. Originally the user model was created with most but not all of its final number of columns. Id, username, first\_name, last\_name, email, password, bio and role were included in the original version of the model. This was due to the nonexistence of the groups, stats, or any connection to the RIOT games or Discord API’s (these would be added at a later stage).

    group\_associations = db.relationship('UserGroup', back\_populates='user', lazy='dynamic')

    groups = db.relationship('Group', secondary='UserGroup', viewonly=True)

The groups column seen above is a many to many relationship, through the group associations column included in the model it associates itself with the UserGroup table which is a linking table between the user and group models. This relationship then back populates with the user information so that when the user information is called within the group it is displayed.

stats = db.relationship('Stats', back\_populates='user', uselist=False)

Under the groups’ relationship columns lies the stats relationship, this is a one to one relationship with the user. This means that the stats are stored in their table with a user foreign key, this foreign key associates the stats information with the user information, this allows the stats information to be called and associated with the user.

    def to\_dict(self):

        return {

            'id': self.id,

            'username': self.username,

            'first\_name': self.first\_name,

            'last\_name': self.last\_name,

            'email': self.email,

            'password': self.password,

            'bio': self.bio,

            'role': self.role,

            'riot\_puuid': self.riot\_puuid,

            'rank': self.rank,

            'riot\_name': self.riot\_name,

            'riot\_tag': self.riot\_tag,

            'discord\_id': self.discord\_id,

            'groups': [group.to\_dict() for group in self.groups],

            'stats': self.stats.to\_dict()

        }

In each model in a Python Flask back end there is a to\_dict() function, this function displays user information that can be displayed in cards, tables or anything needed by the front end. In the user model all of the information is passed to the to\_dict() function as “self” and then displayed through self.{column name}. Originally the user to\_dict() function only included the columns as listed above without the riot information, groups or stats included. The integration and functionality of these will be explained at a later point when discussing the groups, stats, and API integration into the back end.

    def to\_dict\_no\_groups(self):

        return {

            'id': self.id,

            'username': self.username,

            'first\_name': self.first\_name,

            'last\_name': self.last\_name,

            'email': self.email,

            'password': self.password,

            'bio': self.bio,

            'role': self.role,

            'discord\_id': self.discord\_id,

            'riot\_name': self.riot\_name,

            'riot\_tag': self.riot\_tag,

            'riot\_puuid': self.riot\_puuid,

            'rank': self.rank

        }

A to\_dict\_no\_groups() function was included in the user group, this was for when a user may not be involved in any groups in FTM. If that was the case and the original to\_dict() would be used a 404 error would occur and the back end would crash due to there being no such function as “to\_dict()” included in a group that at that point did not exist in an empty array send from the back end. Thus looping through an empty array and attempting to call a “to\_dict()” function would cause an error.

'groups': [group.to\_dict() for group in self.groups]

### Controller & Routes

The user controller is the logic center for how the user information can be accessed, created, deleted, updated and so on. Here I will go through each of the controller functions for the user information.

**Register**

def register():

    email = request.json['email']

    username = request.json['username']

    test\_username = User.query.filter\_by(username=username).first()

    if test\_username:

        return jsonify(message='That username already exists', status=409), 409

    test\_email = User.query.filter\_by(email=email).first()

    if test\_email:

        return jsonify(message='That email is already in use', status=409), 409

    else:

        first\_name = request.json['first\_name']

        last\_name = request.json['last\_name']

        password = request.json['password']

        hashed\_password = generate\_password\_hash(password, method='pbkdf2:sha256')

        role = request.json['role'].capitalize()

        bio = request.json['bio']

        riot\_name = request.json['riot\_name']

        riot\_tag = request.json['riot\_tag']

        riot\_region = request.json['riot\_region'].lower()

        user = User(username=username, first\_name=first\_name, last\_name=last\_name, email=email, password=hashed\_password, role=role, bio=bio, riot\_name=riot\_name, riot\_tag=riot\_tag, riot\_region=riot\_region, riot\_puuid=None, discord\_id=None)

        db.session.add(user)

        db.session.commit()

**The function seen above is the user register function, this function creates the user information an pushes it to the database.**

    email = request.json['email']

    username = request.json['username']

    test\_username = User.query.filter\_by(username=username).first()

    if test\_username:

        return jsonify(message='That username already exists', status=409), 409

    test\_email = User.query.filter\_by(email=email).first()

    if test\_email:

        return jsonify(message='That email is already in use', status=409), 409

**The first section of the function tests weather the username and email address are unique by pulling them from the form and checking them against the database.**

    else:

        first\_name = request.json['first\_name']

        last\_name = request.json['last\_name']

        password = request.json['password']

        hashed\_password = generate\_password\_hash(password, method='pbkdf2:sha256')

        role = request.json['role'].capitalize()

        bio = request.json['bio']

        riot\_name = request.json['riot\_name']

        riot\_tag = request.json['riot\_tag']

        riot\_region = request.json['riot\_region'].lower()

        user = User(username=username, first\_name=first\_name, last\_name=last\_name, email=email, password=hashed\_password, role=role, bio=bio, riot\_name=riot\_name, riot\_tag=riot\_tag, riot\_region=riot\_region, riot\_puuid=None, discord\_id=None)

        db.session.add(user)

**If the username and email pass the test for uniqueness the user information is then created with the information sent through the form, the password is hashed to encrypt it and is decrypted in the login function. riot\_puuid and discord\_id are left as None to be filled in next by accessing their API’s.**

    riot\_url = f"https://europe.api.riotgames.com/riot/account/v1/accounts/by-riot-id/{riot\_name}/{riot\_tag}?api\_key={api\_key}"

    response = requests.get(riot\_url)

    if response.status\_code == 200:

        riot\_data = response.json()

        user.riot\_puuid = riot\_data.get('puuid')

        db.session.commit()

**In the next section of the register function the back end calls the RIOT API with the information previously provided by the user and an API key that is coded into the back end ENV, the users riot\_puuid is then updated.**

    url = f"https://americas.api.riotgames.com/lol/match/v5/matches/by-puuid/{riot\_data.get('puuid')}/ids?start=0&count=20"

    rank\_url = f"https://na1.api.riotgames.com/lol/league/v4/entries/by-puuid/{riot\_data.get('puuid')}"

    rank\_response = requests.get(rank\_url, headers={'X-Riot-Token': api\_key})

    response = requests.get(url, headers={'X-Riot-Token': api\_key})

**Then the users riot matches information is queried and stored inside the back end to be used in the next section.**

    user.rank = rank\_response[0]['tier'] if rank\_response else 'Unranked'

    for match\_id in matches:

        url = f"https://americas.api.riotgames.com/lol/match/v5/matches/{match\_id}"

        response = requests.get(url, headers={'X-Riot-Token': api\_key})

        if response.status\_code != 200:

            return jsonify({'message': 'Error fetching match'}), 500

        match = response.json()

        part\_index = match['metadata']['participants'].index(riot\_data.get('puuid'))

        stats\_list = match['info']['participants'][part\_index]

        kills += stats\_list['kills']

        deaths += stats\_list['deaths']

        assists += stats\_list['assists']

        wins += 1 if stats\_list['win'] else 0

        losses += 0 if stats\_list['win'] else 1

        rank = stats\_list['summonerLevel']

        last20kills.append(stats\_list['kills'])

        last20deaths.append(stats\_list['deaths'])

        last20assists.append(stats\_list['assists'])

    winloss = wins / losses

    kda = (kills + assists) / deaths

    kapm = (kills + assists) / (wins + losses)

    winpercent = int(wins / (wins + losses) \* 100)

    if user.stats:

        user.stats.kills = kills

        user.stats.deaths = deaths

        user.stats.assists = assists

        user.stats.wins = wins

        user.stats.losses = losses

        user.stats.rank = rank

        user.stats.winloss = winloss

        user.stats.kda = kda

        user.stats.kapm = kapm

        user.stats.winpercent = winpercent

        user.stats.last20kills = json.dumps(last20kills)

        user.stats.last20deaths = json.dumps(last20deaths)

        user.stats.last20assists = json.dumps(last20assists)

    else:

        stats = Stats(kills=kills, deaths=deaths, assists=assists, wins=wins, losses=losses, rank=rank, winloss=winloss, kda=kda, kapm=kapm, winpercent=winpercent, last20kills=json.dumps(last20kills), last20deaths=json.dumps(last20deaths), last20assists=json.dumps(last20assists), user=user)

        db.session.add(stats)

    db.session.commit()

    return jsonify(message='User and Riot account linked successfully', status=201), 201

**Once the users PUUID and match ID information had been stored the back end then calls the users last 20 matches (lifetime data is only available with a production API key which is only obtainable from RIOT when creating a production level licensed application) through the users participant index in each match. This is done by cross referencing the user PUUID with all other user PUUID’s in the match and retrieving our users information. This data is then stored in a stats object that is connected to our user through the one to one relation with Stats that each user has.**

**This is then all pushed to the database and the user is then created with their statistics attached.**

**Discord Integration**

def discord\_login():

    discord\_auth\_url = f"https://discord.com/api/oauth2/authorize?client\_id={DISCORD\_CLIENT\_ID}&redirect\_uri={DISCORD\_REDIRECT\_URI}&response\_type=code&scope=identify"

    return redirect(discord\_auth\_url)

@user\_routes.route('/discord\_login', methods=['GET'])

def discord\_login\_route():

    token = request.args.get('token')

    if not token:

        return jsonify({'message': 'Missing token'}), 401

    session['token'] = token

    return discord\_login()

Once the user registers they will then utilize the dicord login route. This route uses FindMyTeams development Discord API information to call to the Discord API, it then returns to the backend and calls the discord authentication URL with the users authentication token, which subsequently links back to the FMT back end and calls the discord callback function.

def discord\_callback(user):

    code = request.args.get('code')

    if not code:

        return jsonify(message='Authorization failed', status=400), 400

    data = {

        'client\_id': DISCORD\_CLIENT\_ID,

        'client\_secret': DISCORD\_CLIENT\_SECRET,

        'grant\_type': 'authorization\_code',

        'code': code,

        'redirect\_uri': DISCORD\_REDIRECT\_URI,

        'scope': 'identify'

    }

    headers = {

        'Content-Type': 'application/x-www-form-urlencoded'

    }

    response = requests.post(f"{DISCORD\_API\_BASE\_URL}/oauth2/token", data=data, headers=headers)

    response\_data = response.json()

    access\_token = response\_data.get('access\_token')

    token\_type = response\_data.get('token\_type')

    headers = {

        'Authorization': f"{token\_type} {access\_token}"

    }

    user\_response = f"{DISCORD\_API\_BASE\_URL}/users/@me"

    user\_response = requests.get(user\_response, headers=headers)

    user\_data = user\_response.json()

    discord\_id = user\_data.get('id')

    user.discord\_id = discord\_id

    db.session.commit()

    return redirect(f"https://fmt-front-end.vercel.app/home")

@user\_routes.route('/discord\_callback', methods=['GET'])

def discord\_callback\_route():

    token = session.get('token')

    if not token:

        return jsonify({'message': 'Missing token'}), 401

    try:

        decoded = decode\_token(token)

        user\_id = decoded['sub']

        user = User.query.get(user\_id)

    except Exception:

        return jsonify({'message': 'Invalid token'}), 401

    if not user:

        return jsonify({'message': 'User not found'}), 404

    return discord\_callback(user)

After the discord callback function is called it executes the route above, which attempts to obtain the user ID from the decoded token sent through the url parameters from the login function then through to the callback function. This is done instead of using @jwt\_required and get\_jwt\_identity() because discord itself does not hold onto the JWT token sent through headers once the callback function is called. This means that the token must be sent through the parameters otherwise obtaining the users information is not possible.

Before the user identity is actually obtained discord itself creates a data response and sends it back to the FMT back end. Once that is done the user identity is found, then updated with the discord id just obtained from the discord API.

**Login**

def login():

    if request.is\_json:

        email = request.json['email']

        password = request.json['password']

    else:

        email = request.form['email']

        password = request.form['password']

    user = User.query.filter\_by(email=email).first()

    if user and check\_password\_hash(user.password, password):

        access\_token = create\_access\_token(identity=str(user.id))

        session['id'] = user.id

        return jsonify(message='Login succeeded!', access\_token=access\_token)

    else:

        return jsonify(message='Invalid email or password', status=401), 401

The user login function checks if the form is json or a form field, then queries through the Users in the database to confirm that user exists. Once doing that the password is decrypted and checked against the forms password to make sure they match. If those are successful an access token is created and the user is logged in. The user id is stored in session to help with discord user identification and is broken once a user is successfully logged in.

**Get User/Users**

@user\_routes.route('/profile', methods=['GET'])

@jwt\_required()

def get\_this\_user():

    user\_id = get\_jwt\_identity()

    if not user\_id:

        return jsonify({'message': 'User not found'}), 404

    user = User.query.filter\_by(id=user\_id).first()

    if not user:

        return jsonify({'message': 'User not found'}), 404

    return get\_user(user)

@user\_routes.route('/users', methods=['POST'])

@jwt\_required()

def get\_users\_route():

    user\_id = get\_jwt\_identity()

    if not user\_id:

        return jsonify({'message': 'User not found'}), 404

    user = User.query.filter\_by(id=user\_id).first()

    if not user:

        return jsonify({'message': 'User not found'}), 404

    return get\_users(user)

Both routes for obtaining a single and multiple users require JWT authentication and query the JWT identity to return correct user information. The single user query uses it to define the user to display, whereas the multiple user query uses it to make sure the identified user is NOT returned in the stack.

def get\_user(user):

    return jsonify(user.to\_dict())

def get\_users(user):

    data = request.get\_json()

    liked\_users = db.session.query(Match.liked\_id).filter(Match.liker\_id == user.id)

    liked\_me = db.session.query(Match.liker\_id).filter(Match.liked\_id == user.id)

    query = User.query

    # Apply filters dynamically

    if "role" in data:

        query = query.filter(User.role == data["role"])

    if "rank" in data:

        query = query.filter(User.rank == data["rank"])

    excluded\_ids = set([user.id])

    excluded\_ids.update([row[0] for row in liked\_users])

    excluded\_ids.update([row[0] for row in liked\_me])

    users\_list = query.filter(~User.id.in\_(excluded\_ids)).all()

    result = [user.to\_dict() for user in users\_list]

    return jsonify(result)

In the get\_user() function a simple user.to\_dict() is returned.

For the get\_users() section filters are added in to the request and certain types of users are excluded from the response. The filters section takes in a json file of filters, it then checks if “role” or “rank” are included in the filters, if not no filters are selected and all users are returned.

Users can be excluded from the return for three reasons, either they’re already a liked user, they have liked the currently active user, or they are the currently active user. This list is then looped through and a to\_dict() is returned for each.

**Add User To Group**

def add\_user\_to\_group():

    # Get the data from the request (user\_id and group\_id)

    user\_id = request.json.get('user\_id')

    group\_id = request.json.get('group\_id')

    # Find the user and the group

    user = User.query.get(user\_id)

    group = Group.query.get(group\_id)

    if not user or not group:

        return jsonify({'message': 'User or Group not found'}), 404

    # Check if the link already exists

    existing\_link = UserGroup.query.filter\_by(user\_id=user.id, group\_id=group.id).first()

    if existing\_link:

        return jsonify({'message': 'User already in group'}), 400

    # Create a new UserGroup link

    user\_group = UserGroup(user\_id=user.id, group\_id=group.id)

    db.session.add(user\_group)

    db.session.commit()

    return jsonify({'message': 'User added to group successfully'}), 200

In the add user to group route a user and group are queried, similar to other routes jwt is used to identify the currently active user. Once both are identified it is checked if a link exists, this makes it so a user cannot be added to a group twice. If not a link is created between the user and the group, hence adding the user to the group.

**Update**

def update\_user(user\_id: int):

    user = User.query.filter\_by(id=user\_id).first()

    if user:

        user.username = request.form['username']

        user.first\_name = request.form['first\_name']

        user.last\_name = request.form['last\_name']

        user.bio = request.form['bio']

        user.role = request.form['role']

        user.email = request.form['email']

        user.password = request.form['password']

        db.session.commit()

        return jsonify(message='You updated a profile', status=202), 202

    else:

        return jsonify(message='That profile does not exist', status=404), 404

Here a simple user update function takes form data and updates the user object accordingly.

**Delete**

def delete\_user(user\_id: int):

    user = User.query.filter\_by(user\_id=user\_id).first()

    if user:

        db.session.delete(user)

        db.session.commit()

        return jsonify(message='You deleted your profile', status=202), 202

    else:

        return jsonify(message='That profile does not exist', status=404), 404

In the delete function, a user is queried after their ID is sent as a parameter, once this happens their account is deleted.

## Group & UserGroup

### Model

class Group(db.Model):

    \_\_tablename\_\_ = 'group'

    id = Column(Integer, primary\_key=True)

    group\_name = Column(String)

    hidden = Column(Boolean, default=False)

    user\_associations = db.relationship('UserGroup', back\_populates='group', lazy='dynamic')

    users = db.relationship('User', secondary='UserGroup', viewonly=True)

    def to\_dict(self):

        return {

            'id': self.id,

            'group\_name': self.group\_name,

            'users': [user.to\_dict\_no\_groups() for user in self.users]

        }

The group model is much simpler than the user model. The group requires less data to be stored, importantly users are shown within the group through their user\_association and users columns which back populate the user and group through their UserGroup model. The hidden column is used for a soft delete of group.

class UserGroup(db.Model):

    \_\_tablename\_\_ = 'UserGroup'

    user\_id = db.Column(db.Integer, db.ForeignKey('user.id'), primary\_key=True)

    group\_id = db.Column(db.Integer, db.ForeignKey('group.id'), primary\_key=True)

    seen = db.Column(db.Boolean, default=False)

    user = db.relationship('User', back\_populates='group\_associations')

    group = db.relationship('Group', back\_populates='user\_associations')

The UserGroup model stores the user id and the group id included in the link to allow them to reference each other. Also stored in the model is a seen column, this is used to handle new groups and notifications in the front end.

### Controller & Routes

**Create Group**

def create\_group(user):

    data = request.get\_json()

    group\_name = data.get('group\_name')

    if not group\_name:

        return jsonify({'message': 'Group name is required'}), 400

    # Create group and add creator

    group = Group(group\_name=group\_name)

    db.session.add(group)

    db.session.flush()

    user\_group = UserGroup(user\_id=user.id, group\_id=group.id, seen=False)

    db.session.add(user\_group)

    db.session.commit()

    return jsonify({'message': 'Group created!', 'group': group.to\_dict()}), 201

In the create group function a json form is sent to the back end, the group name is taken from the form then the group is created, once created a db.flush() is executed which allows a UserGroup connection to be made which adds the currently active user to the group.

**Get Groups**

def get\_groups(user):

    groups = []

    for association in user.group\_associations:

        group = association.group

        if group and not group.hidden:

            group\_data = group.to\_dict()

            group\_data['seen'] = association.seen

            groups.append(group\_data)

    if not groups:

        return jsonify({'message': 'No groups found'}), 404

    return jsonify(groups), 200

Here groups are queried by using the currently active user (jwt get identity used in the route as shown previously) and checking that the groups are not actively hidden. Once a group is confirmed a group\_data variable is created, the group.to\_dict() is added to it, then the seen tag is queried from the UserGroup association table and added to the group data. Upon completion of this the groups are returned.

**Get Group**

def get\_group\_by\_id(group\_id, user):

    group = Group.query.get(group\_id)

    if not group:

        return jsonify({'message': 'Group not found'}), 404

    assoc = UserGroup.query.filter\_by(user\_id=user.id, group\_id=group.id).first()

    if not assoc:

        return jsonify({'message': 'User not in group'}), 403

    assoc.seen = True

    db.session.commit()

    return jsonify(group.to\_dict()), 200

In the get single group function, the group is queried using the group id, also queried is the user. The group and user are queried to check if the user is included in the group, if so the seen column in the UserGroup connection is changed to True, meaning the group is no longer new. The group is then returned.

**Hide Group (Soft Delete)**

def hide\_group(group\_id):

    group = Group.query.get(group\_id)

    if not group:

        return jsonify({'message': 'Group not found'}), 404

    group.hidden = True

    db.session.commit()

    return jsonify({'message': 'Group hidden successfully!'}), 200

The group in question in the hide group route is queried through the group\_id sent in the parameters of the URL, the group is then hidden.

**Check New Groups**

def check\_new\_groups(user):

    unseen\_groups = (db.session.query(UserGroup).filter\_by(user\_id=user.id, seen=False).all())

    if not unseen\_groups:

        return jsonify({'message': 'No new groups found'}), 200

    return jsonify([group.group.to\_dict() for group in unseen\_groups]), 200

The check new groups route is a route designed to check the back end and database for new groups that the current user is involved in. This route is polled by the front end to allow notifications to work.

## Matches

### Model

class Match(db.Model):

    \_\_tablename\_\_ = 'match'

    id = Column(Integer, primary\_key=True)

    liker\_id = Column(Integer, ForeignKey('user.id'), nullable=False)

    liked\_id = Column(Integer, ForeignKey('user.id'), nullable=False)

    status = Column(String, default="pending") #Other values are "accepted" and "rejected"

    timestamp = Column(DateTime, default=datetime.now)

    liker = db.relationship('User', foreign\_keys=[liker\_id], backref='likes\_sent')

    liked = db.relationship('User', foreign\_keys=[liked\_id], backref='likes\_received')

    def to\_dict(self):

        return {

            'id': self.id,

            'liker\_id': self.liker\_id,

            'liked\_id': self.liked\_id,

            'status': self.status,

            'timestamp': self.timestamp,

            'liker': self.liker.to\_dict(),

            'liked': self.liked.to\_dict()

        }

The match is a one to many relationship between users and matches, this is made so because each match has a liker and a liked user, shown above both as foreign keys relating back to the user associated with each. This is done to allow querying of likes sent vs likes received and to simplify the back end and nullifying the need for another linking table like with user and group.

### Controller & Routes

**Send Like**

def send\_like(liked\_id, user):

    existmatch = Match.query.filter\_by(liker\_id=user.id, liked\_id=liked\_id).first()

    if existmatch:

        return jsonify({'message': 'You liked this user already'}), 400

    match = Match(liker\_id=user.id, liked\_id=liked\_id)

    db.session.add(match)

    db.session.commit()

    return jsonify({'message': f"Like sent by: {user.id}"}), 201

The send like route uses the liked\_id, gotten from the URL parameters and the current user, gotten from a JWT identity check. It uses these two items to firstly check if a prior match exists between these users, if not a new one is created.

**Likes Sent**

def get\_likes(user):

    matches = Match.query.filter\_by(liker\_id=user.id, status="pending").all()

    result = [match.to\_dict() for match in matches]

    return jsonify(result), 200

To attain the likes sent by the current user, their identity is gathered and matches are queried based on that users id as the id of the LIKER. A to\_dict() for each match is then returned.

**Likes Received**

def get\_my\_likes(user):

    matches = Match.query.filter\_by(liked\_id=user.id, status="pending").all()

    result = [match.to\_dict() for match in matches]

    return jsonify(result), 200

Likes received is the same as likes sent but the users id is queried as the LIKED id in place of the liker.

**Accept Match**

def accept\_match(match\_id):

    match = Match.query.filter\_by(id=match\_id, status="pending").first()

    if not match:

        return jsonify({'message': 'No like found'}), 404

    match.status = "accepted"

    db.session.commit()

    return jsonify({'message': 'Match accepted'}), 200

To accept a match the user calls the accept match route, which is queried using the match id gathered from the URL parameters. This matches status is then updated to accepted through a put request.

**Decline Match**

def decline\_match(match\_id):

    match = Match.query.filter\_by(id=match\_id, status="pending").first()

    if not match:

        return jsonify({'message': 'No like found'}), 404

    match.status = "rejected"

    db.session.commit()

    return jsonify({'message': 'Match declined'}), 200

Declining match is similar to accepting a match, but rather than changing the status to “accept” it is changed to “rejected”.

**Get Friends**

def get\_friends(user):

    matches = Match.query.filter(

        or\_(

            Match.liker\_id == user.id,

            Match.liked\_id == user.id

        ),

        Match.status == "accepted"

    ).all()

    result = []

    for match in matches:

        data = match.to\_dict()

        data['current\_user'] = user.id

        result.append(data)

    return jsonify(result), 200

The get friends route queries the user identity using JWT, then checks if that user is included in any match where the user is either liker or liked and the status of the match is “accepted”, if these criteria are met each match is added to a result array along with the current user. This is to allow the front end to check which user is which within the match and display the opposite as a friend through the friend card in the friends list.

**Star Users**

def send\_star(liked\_id, user):

    existmatch = Match.query.filter\_by(liker\_id=user.id, liked\_id=liked\_id).first()

    if existmatch:

        return jsonify({'message': 'You liked this user already'}), 400

    match = Match(liker\_id=user.id, liked\_id=liked\_id, status="starred")

    db.session.add(match)

    db.session.commit()

    return jsonify({'message': f"Star sent by: {user.id}"}), 201

Starring users is another form of matching, but is used when a user is unsure if they want to like a user or not, in the starred users tab the user can like or undo a like using the like button and decline button, calling each respective route.

**Get Starred**

def get\_starred(user):

    matches = Match.query.filter\_by(liker\_id=user.id, status="starred").all()

    result = [match.to\_dict() for match in matches]

    return jsonify(result), 200

Get starred is almost identical to get friends.

**Change Match to Like**

def change\_match(match\_id):

    match = Match.query.filter\_by(id=match\_id, status="starred").first()

    if not match:

        return jsonify({'message': 'No like found'}), 404

    match.status = "pending"

    db.session.commit()

    return jsonify({'message': 'Like Sent'}), 200

The change match route is similar to the create like route, except for setting the status to starred instead of the default “pending” value.

## Stats

### Model

class Stats(db.Model):

    \_\_tablename\_\_ = 'stats'

    id = Column(Integer, primary\_key=True)

    kills = Column(Integer)

    deaths = Column(Integer)

    assists = Column(Integer)

    wins = Column(Integer)

    losses = Column(Integer)

    rank = Column(String)

    winloss = Column(Float)

    kda = Column(Float)

    kapm = Column(Float)

    winpercent = Column(Integer)

    last20kills = Column(String)

    last20deaths = Column(String)

    last20assists = Column(String)

    user\_id = db.Column(db.Integer, db.ForeignKey('user.id'), unique=True)

    user = db.relationship('User', back\_populates='stats')

    def to\_dict(self):

        return {

            'kills': self.kills,

            'deaths': self.deaths,

            'assists': self.assists,

            'wins': self.wins,

            'losses': self.losses,

            'rank': self.rank,

            'winloss': self.winloss,

            'kda': self.kda,

            'kapm': self.kapm,

            'winpercent': self.winpercent,

            'last20kills': json.loads(self.last20kills) if self.last20kills else [],

            'last20deaths': json.loads(self.last20deaths) if self.last20deaths else [],

            'last20assists': json.loads(self.last20assists) if self.last20assists else [],

        }

The stats model stores all the statistical data attained from the RIOT Games API. There are a number of statistics including three arrays storing each matches kills, deaths and assists to be utilised in graphs on the front end.

The stats section does not include any controller or routes as it is directly related to and only queried through the user model, controller and routes.

# Front End & UI Development & Implementation

## Overview

Front end development was handled using React-Vite and was subsequently hosted on Vercel. Development took place throughout the sprint weeks and was roughly half the time spent on the application development.

User Interface design was developed utilizing Shadcn and Tailwind for styling. Shadcn is a Tailwind library where the developer can import specific components from the library using npx install commands in the CLI. This makes it incredibly easy to design a user interface which looks good and performs for each user easily.

The front end implementation section will be split into pages as they are on FindMyTeam. Each component on these pages will be explored, what they do, how they were developed and how they were implemented within FMT.

## Contexts, Configs, Router and Navigation

**API Config**

import axios from 'axios';

const local = axios.create({

    baseURL: 'https://fmtbackendpy.onrender.com'

});

export default [local];

The API config is used to store the API URL and serve it to each page upon import. The reason for this config is to allow a change in API URL to be handled without having to manually reset or change ever call on every page of the application.

**AuthContext**

export function useAuth () {

    const context = useContext(AuthContext);

    return context;

}

export const AuthProvider = ({ children }) => {

    const [token, setToken] = useState(localStorage.getItem("token") || null);

    const [authenticated, setAuthenticated] = useState(false);

    useEffect(() => {

        if (token) {

            setAuthenticated(true);

            localStorage.setItem("token", token);

        }else {

            setAuthenticated(false);

            localStorage.removeItem("token");

        }

    },[token]);

    const login = async (email, password) => {

        try {

            const response = await local.post("/login", { email, password });

            setToken(response.data.access\_token);

            return response.data;

        } catch (error) {

            throw error.response.data;

        }

    };

    const logout = () => {

        setToken(null);

    };

    return(

        <AuthContext.Provider value={{ authenticated, token, login, logout,

            onAuthenticated: (auth, token) => {

                setAuthenticated(auth);

                if (auth && token) {

                    localStorage.setItem('token', token);

                } else{

                    localStorage.removeItem('token');

                }

            }

        }}>

            {children}

        </AuthContext.Provider>

    )

}

The AuthContext provides authentication context to the entire website, this return is wrapped around the <App/> import in the index.jsx for the front end, as seen below.

    <React.StrictMode>

        <AuthProvider>

            <SidebarProvider>

                <App />

            </SidebarProvider>

        </AuthProvider>

    </React.StrictMode>

**Router (App.jsx)**

    if(authenticated){

            protectedRoutes = (

            <>

            <Route path='/social' element={

                <PrivateRoute>

                    <SocialPage/>

                </PrivateRoute>

                }/>

            <Route path='/groups/:group\_id' element={

                <PrivateRoute>

                    <GroupPage/>

                </PrivateRoute>

                }/>

            <Route path='/add\_user/:group\_id' element={

                <PrivateRoute>

                    <AddUserPage/>

                </PrivateRoute>

                }/>

            <Route path='/profile' element={

                <PrivateRoute>

                    <UserPage/>

                </PrivateRoute>

                }/>

            <Route path="/home" element={

                <PrivateRoute>

                    <MainPage/>

                </PrivateRoute>

                }/>

                <Route path="/help" element={

                <PrivateRoute>

                    <HelpPage/>

                </PrivateRoute>

                }/>

            </>

            );

    }

    return (

        <Router>

            <GroupNotificationWatcher />

            <LocationAwareLayout />

            <Toaster position="top-center" toastOptions={{ className: 'sonner-toast', duration: 4000 }} />

            <Routes>

                {protectedRoutes}

                <Route path='/' element={<WelcomePage />} />

                <Route path='/login' element={<LoginPage />} />

                <Route path='/register' element={<RegisterPage />} />

                <Route path='/firstlogin' element={<FirstTimeLoginForm />} />

                <Route path ='\*' element={<PageNotFoundPage />} />

            </Routes>

        </Router>

The router handles all navigation routing for FindMyTeam, React-Router was used for its construction. In the router there are protected and non-protected routes, if the user is authenticated the protected routes are available to be accessed, otherwise only the non-protected routes are available.

    useEffect(() => {

        if(localStorage.getItem('token')){

        onAuthenticated(true);

        }

        setTimeout(() => {

            setLoading(false);

        }, 100);

    }, []);

The authentication is gathered upon page mount by this useEffect. The private route function directs the user to the login page if that user is not authenticated.

**Navigation (Sidebar)**

    const items = [

    {

        title: "Home",

        url: "/home",

        icon: Home,

    },

    {

        title: "Social",

        url: "/social",

        icon: Inbox,

    },

    {

        title: "Profile",

        url: "/profile",

        icon: UserSquare2Icon,

    },

    {

        title: "Help",

        url: "/help",

        icon: HelpCircleIcon,

    }

    ]

    export function AppSidebar() {

    const hasNewGroup = GroupNotifications()

    return (

        <Sidebar>

        <SidebarContent>

            <SidebarGroup>

            <SidebarGroupLabel className="text-primary font-bold">FindMyTeam</SidebarGroupLabel>

            <SidebarGroupContent>

                <SidebarMenu className="flex flex-col justify-between h-full">

                {items.map((item) => (

                    <SidebarMenuItem key={item.title}>

                    <SidebarMenuButton asChild>

                        <a href={item.url}>

                        <item.icon />

                        <span>{item.title}</span>

                        {item.title === "Social" && hasNewGroup && (

                        <span className="absolute -top-1 -right-1 h-2 w-2 rounded-full bg-red-500"></span>

                        )}

                        </a>

                    </SidebarMenuButton>

                    </SidebarMenuItem>

                ))}

                </SidebarMenu>

            </SidebarGroupContent>

            </SidebarGroup>

        </SidebarContent>

                <div className="p-2 w-full flex flex-col gap-2">

                <LogoutButton className="w-full"/>

                </div>

        </Sidebar>

Navigation on FMT is handled using a Shadcn sidebar which is wrapped around the <App/> component in the index.jsx page of the application. This is the same as the authprovider and can be seen in the code snippet showing such previously.

In the sidebar the items to be shown are stored in an array of objects, these are then mapped to the sidebar through a .map foreach function.

In the sidebar a hasNewGroup function is shown, this function determines whether a user is in possession of an unopened or unseen group and displays a notification dot above the social button if the user does have any new groups. (This function will be expanded on later when discussing the social page)

## Register & Login

**Register**

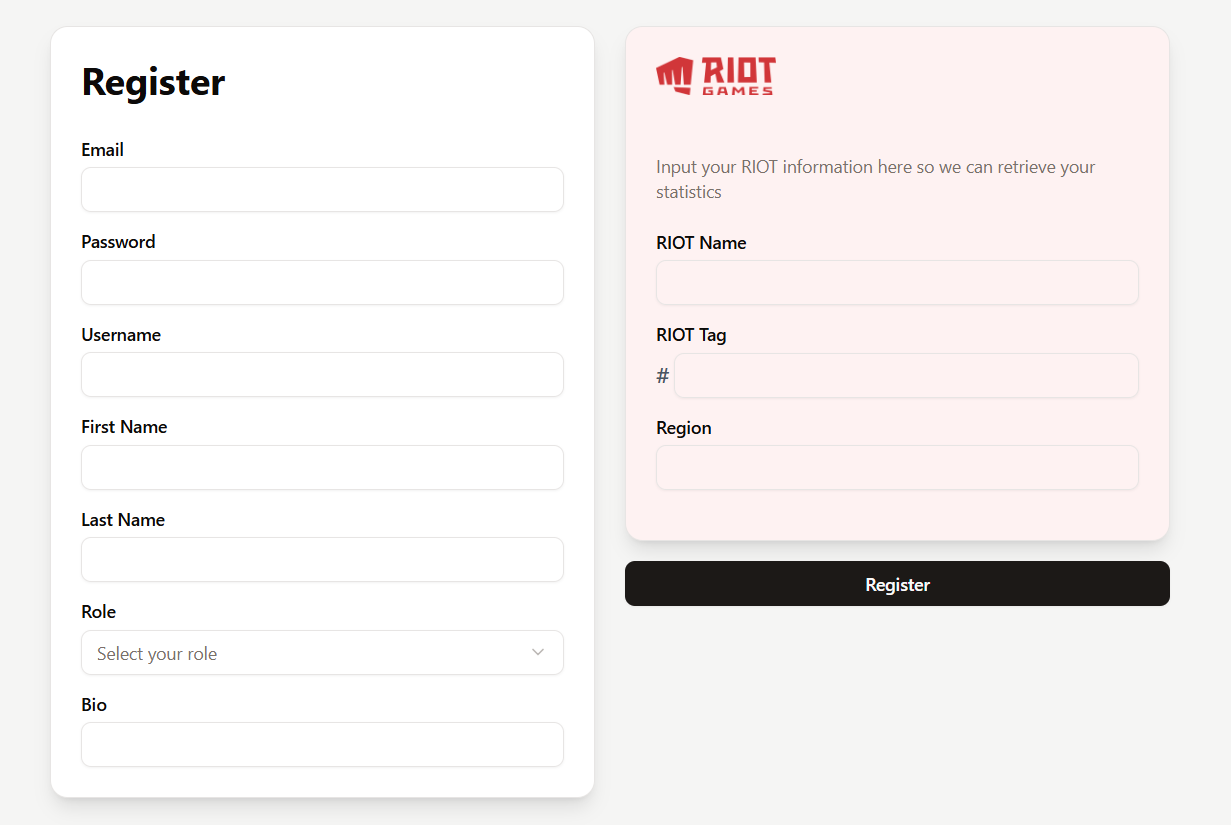


Figure 8.2.1 Register page for FMT.

On the register page the user is prompted to input their information to create an account within FindMyTeam. Above seen are the form fields and ShadCN components used to create the page.

These Shadcn components include the Card and related components, used for both individual input sections, Select and related, used for the selection of role, Label, Button and Input all used throughout the page.

const RegisterForm = () => {

    const [local] = axios;

    const [loading, setLoading] = useState(false);

    const navigate = useNavigate();

    const [form, setForm] = useState({

        email: "",

        password: "",

        username : "",

        first\_name : "",

        last\_name : "",

        role: "",

        bio: "",

        riot\_name:"",

        riot\_tag:"",

        riot\_region:"",

    });

    const handleClick = () => {

        setLoading(true);

        // let regToken = localStorage.getItem('token');

        local.post('/register', {

            email: form.email,

            password: form.password,

            username : form.username,

            first\_name : form.first\_name,

            last\_name : form.last\_name,

            role: form.role,

            bio: form.bio,

            riot\_name: form.riot\_name,

            riot\_tag: form.riot\_tag,

            riot\_region: form.riot\_region

        })

        .then(response => {

            console.log(response);

            navigate('/firstlogin', {state: { email: form.email, password: form.password }});

        })

        .catch(err => {

            console.error(err);

            toast.error(

                err?.response?.data?.message || "An unexpected error occurred."

            );

        })

        .finally(() => {

            setLoading(false);

        });

    }

    const handleKeyDown = (e) => {

        if (e.key === "Enter") {

            handleClick();

        }

    };

    const handleForm = (e) => {

        setForm(prevState => ({

            ...prevState,

            [e.target.name]: e.target.value

        }));

    };

Within the register page useStates are used to hold information such as loading, form state, and then to navigate to the discord login connection page with state data to be used to reduce the likelihood of a user inputting wrong information on the discord login page.

Local.post is used in the handleclick() function to post the information gained from the form sections within the front end inputted by the user to the back end where the user is registered.

A Shadcn toast is displayed in the case of any errors occurring in the register form, which are sent by the back end.

A setLoading() function is also used to display a spinner (below) which will be common on many pages moving forward, this is to communicate to the user that FMT is currently loading information and not to worry or click off.

    if (loading) {

        return (

            <div className="flex justify-center items-center min-h-full w-full bg-secondary">

                <div className="w-16 h-16 border-4 border-t-transparent border-primary rounded-full animate-spin" />

            </div>

        );

    }

**First Time Login & Discord Integration**

const FirstTimeLoginForm = () => {

    const location = useLocation();

    const { email, password } = location.state || { };

    const [local] = axios;

    const { onAuthenticated } = useAuth();

    const errorStyle = {

        color: 'red'

    };

    const [error, setError] = useState(null);

    const [form, setForm] = useState({

        email: email || "",

        password: password || ""

    });

    const [errorMessage, setErrorMessage] = useState("");

    const handleDiscord = (token) => {

        window.location.href = `https://fmtbackendpy.onrender.com/discord\_login?token=${token}`;

    }

    const handleClick = () => {

        // let regToken = localStorage.getItem('token');

        local.post('/login', {

            email: form.email,

            password: form.password

        })

        .then(response => {

            console.log(response);

            const  token  = response.data.access\_token;

            onAuthenticated(true, token);

            handleDiscord(response.data.access\_token);

        })

        .catch(err => {

            console.error(err);

            setErrorMessage(err.response.msg);

        });

    }

The first time login is designed to be accessed only once by each user, although the page is accessible through the URL and can the form can be redone and a new or the same discord ID attached with no adverse consequences.

In the function the email and password are pulled as a state from the register page using the useLocation() function. This allows the form to be filled in with the information attained from the registration of the user if coming from that page in two disabled input locations. This is done to make sure the user inputs the information for the new account they’ve created and not a separate account.

        <CardContent className="space-y-4">

            <div className="space-y-2">

            <Label htmlFor="email">Email</Label>

            <Input

                id="email"

                type="text"

                name="email"

                value={form.email}

                onChange={handleForm}

                onKeyDown={handleKeyDown}

                placeholder="you@example.com"

                disabled={!!email} // Disable if email is provided in state

            />

            </div>

            <div className="space-y-2">

            <Label htmlFor="password">Password</Label>

            <Input

                id="password"

                type="password"

                name="password"

                value={form.password}

                onChange={handleForm}

                onKeyDown={handleKeyDown}

                placeholder="••••••••"

                disabled={!!password} // Disable if password is provided in state

            />

            </div>

            <Button className="w-full extra hover:opacity-80" onClick={handleClick}>

            Login

            </Button>

The code above shows the inputs disabled when the password and email fields are present from a previous page.

The user is routed through the back ends discord login and discord callback functions to attach the discord id to the user.

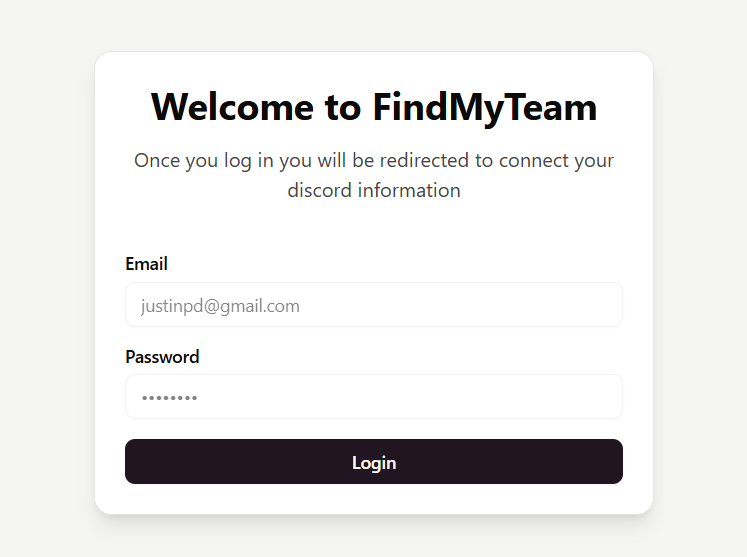


Figure 8.3.1 FindMyTeam first time login page.

Seen above the Shadcn card, input and button components were used to make the first time login form, when information is passed from the register page the inputs are disabled to allow the user to directly click login with ease.

**Login**

const LoginForm = () => {

    const [local] = axios;

    const { onAuthenticated } = useAuth();

    const navigate = useNavigate();

    const [form, setForm] = useState({

        email: "",

        password: ""

    });

    const handleClick = () => {

        // let regToken = localStorage.getItem('token');

        local.post('/login', {

            email: form.email,

            password: form.password

        })

        .then(response => {

            console.log(response);

            const  token  = response.data.access\_token;

            onAuthenticated(true, token);

            navigate('/home');

        })

        .catch(err => {

            console.error(err);

            toast.error(

                err?.response?.data?.message || "An unexpected error occurred."

            );

        });

    }

The login form is very similar to the first time login form for FMT, the main difference being that the main login form does not carry previous data through state from the register page, as it is not accessed through that page. Instead the login form only executes the filling in of the login form and sending a post request to the back end to log said user in.

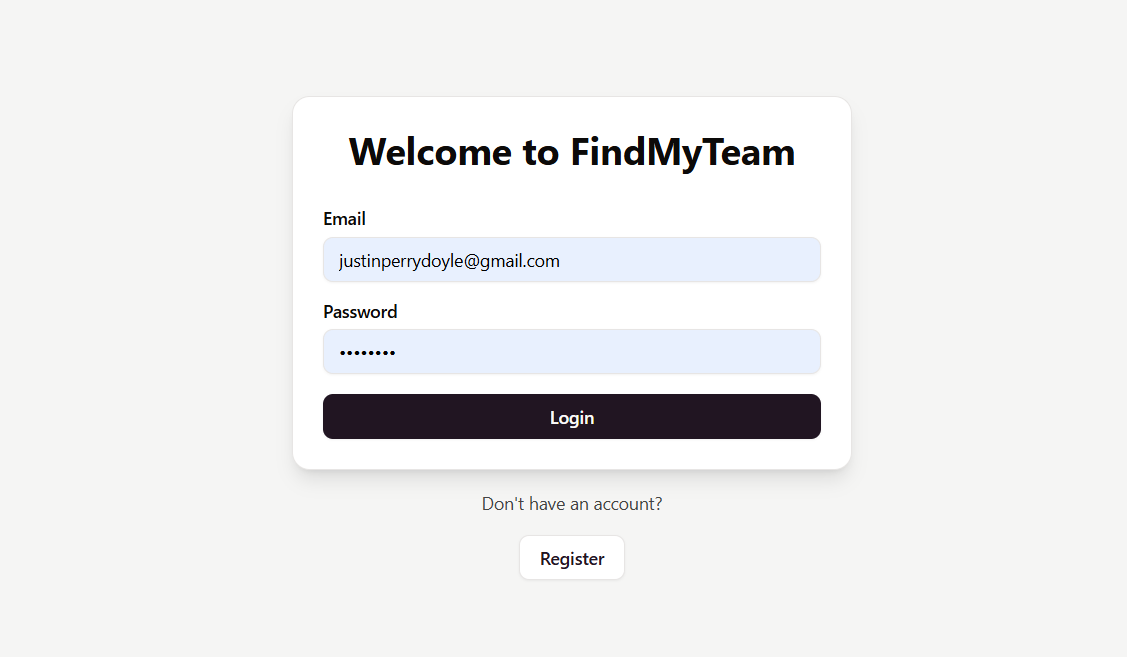


Figure 8.3.2 FindMyTeam login page.

The FMT login page uses nearly identical components from Shadcn as the first time login page.

## Home Page

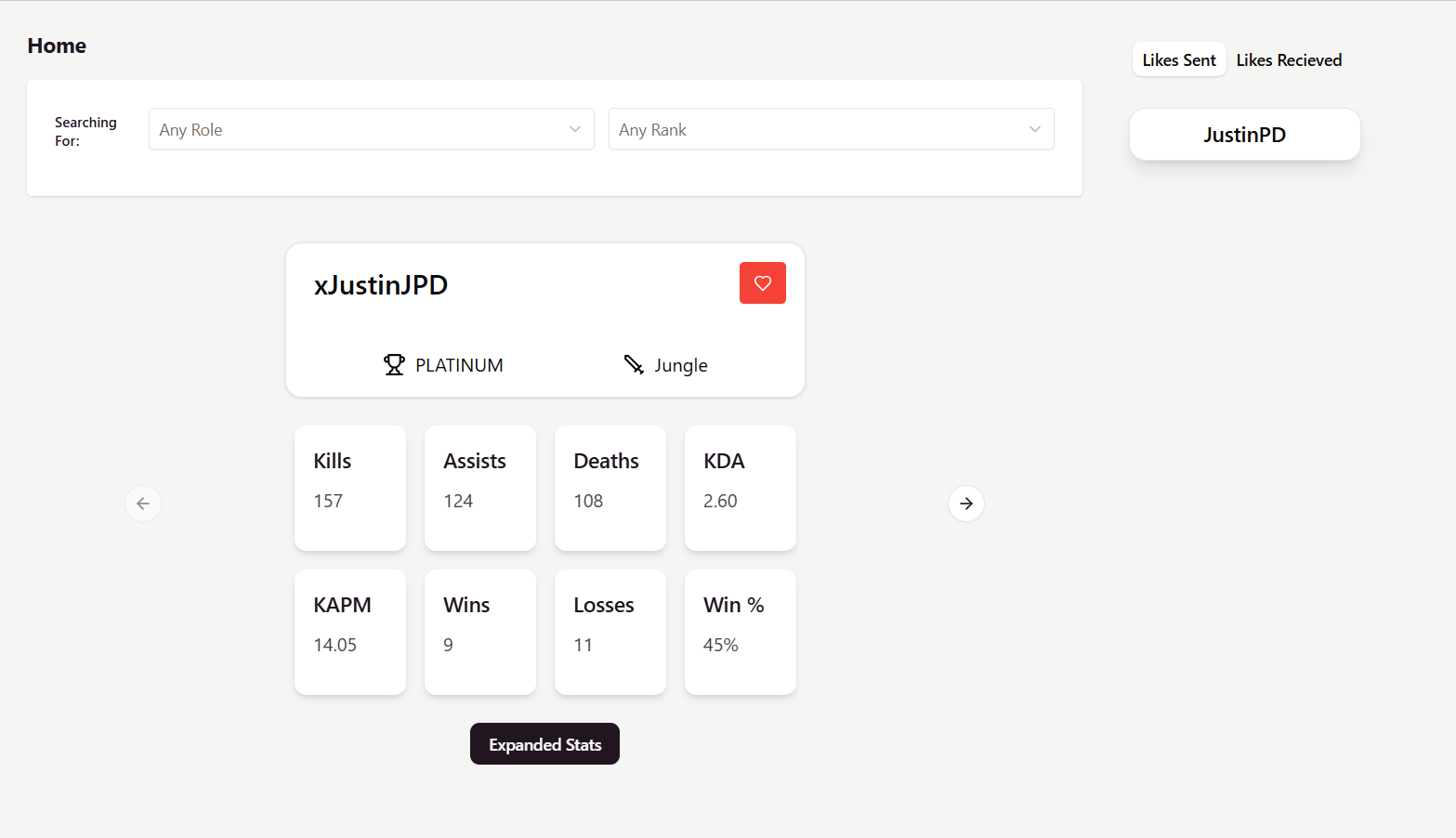


Figure 8.4.1 Home Page for FindMyTeam.

There are multiple components on the FMT homepage, in this section each will be discussed to bring attention to the functionality and design for each.

**Filters**

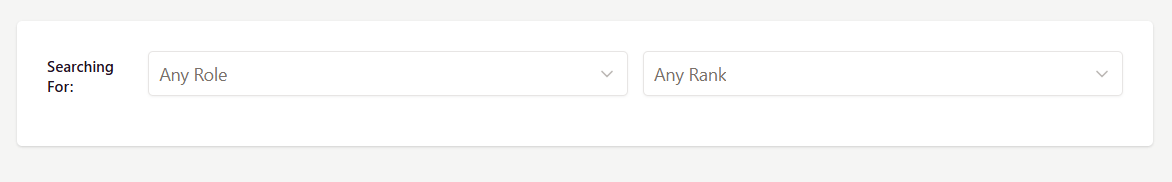


Figure 8.4.2 The filters component found on FMT’s homepage.

The filters component on FMT is designed to allow users to change their preferences when searching for users contained in the user card stack.

Preferences are role and rank based, giving users the freedom to search within any or all ranks and for a specific or non-specified role. This gives users control over who they find within their search stack which is important as in League Of Legends users can only play with other users ranked within one rank of their own. (e.g. a user ranked in GOLD may only play with users ranked within Silver, Gold or Platinum and no others.)

    const FiltersDropdown = ({ onChange }) => {

        const [filters, setFilters] = useState({});

        const handleChange = (e) => {

            const { name, value } = e.target;

            // Check if filter exists

            const updatedFilters = { ...filters };

            if (value === "") {

            delete updatedFilters[name];

            } else {

            updatedFilters[name] = value;

            }

            setFilters(updatedFilters);

            onChange(updatedFilters);

        };

The filters dropdown menus check if filters exist and are selected within the component, if filters are selected the component updates the filters that are sent with the users get request within the search stack, this sets the filters in the back end and returns only the users that meet the filtered criteria.

The filters component uses a card component and two select dropdowns from Shadcn to display this functionality and information to the user.

**User Search Stack Card**

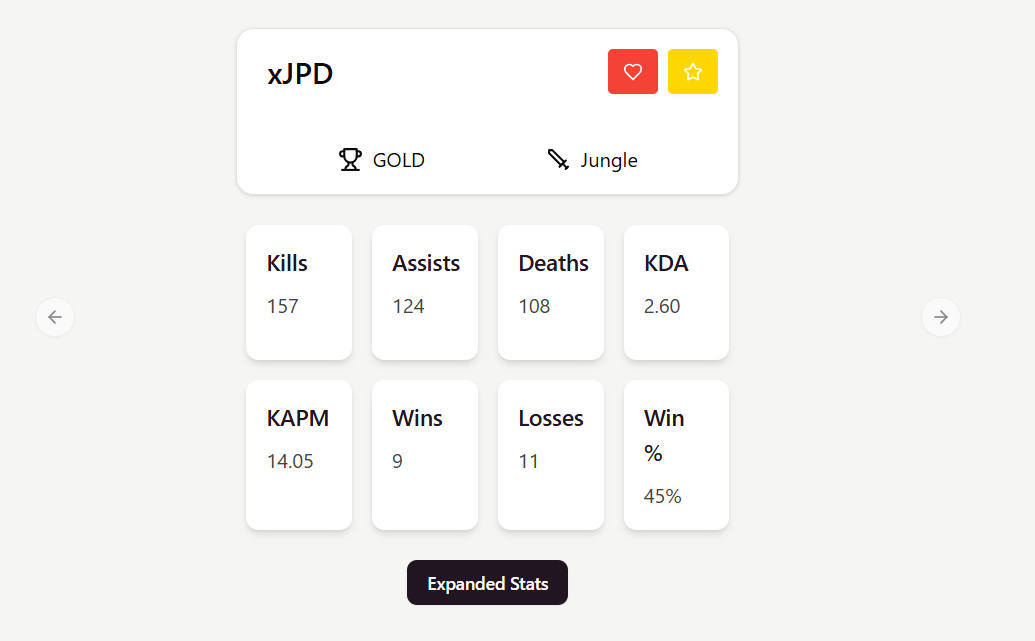


Figure 8.4.3 The user stack card displayed on the home page.

The user card is designed to display important user information to the currently active and searching user. It displays all of the users key information like username, rank, role and statistics to the searching user. These are displayed within small card objects from Shadcn.

Also seen in the user card above is a small heart or “like” button, this button allows the searching user to “like” or “match” with the user who is having their information displayed in the card.

            <Carousel className="justify-items-center items-center w-full ">

                <CarouselContent className="w-full justify-items-center items-center max-w-lg">

                {Array.isArray(users) && users.length > 0 ? (

                    users.map((user) => (

                    <CarouselItem key={user.id}>

                        <UserCard key={user.id} user={user} onLikeSent={handleLike} />

                    </CarouselItem>

                    ))

                ) : (

                    <p className="m-6">No users found.</p>

                )}

                </CarouselContent>

                <CarouselPrevious className="lg:absolute left-25 -translate-x-1/2" />

                <CarouselNext className="lg:absolute right-25 translate-x-1/2" />

            </Carousel>

The user card is stored in a Shadcn carousel, this allows searching users to move between users and “pass” users they would not be interested in matching with.

    const fetchUsers = () => {

        local.post("/users", filters, {

            headers: {

                Authorization: `Bearer ${localStorage.getItem("token")}`,

            },

        })

        .then((response) => {

            const shuffled = [...response.data].sort(() => Math.random() - 0.5);

            setUserList(shuffled);

            setLoading(false);

        })

        .catch((err) => {

            console.error(err);

            if (err.response?.data?.message) {

                setError(err.response.data.message);

            }

        })

        .finally(() => {

            setLoading(false);

        });

    };

    useEffect(() => {

        fetchUsers();

    }, [filters]);

The user information is passed to the card through a mapping function after the information has been gained through the “/users” call to the back end.

Also seen under the fetch users function is a useEffect which calls said fetch users function, this is to allow component loading for the user card stack whenever a user is liked by the searching user, negating the need for the user to refresh the page manually to update both the user card stack and/or the likes sent component.

**Expanded Stats Drawer**

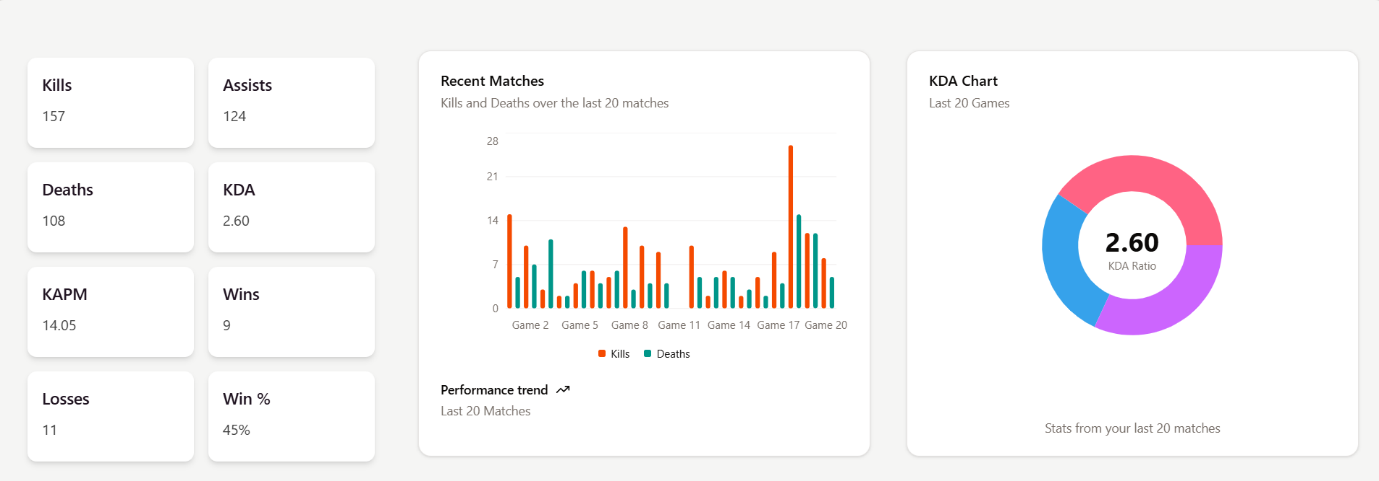


Figure 8.4.3 The expanded stats drawer for the user.

The user statistics drawer is a Shadcn drawer component that displays the users statistics in multiple forms to the user. This drawer is activated through the “expand statistics” button on the home page or by clicking on a users’ name on either the friends list or likes received list on the homepage and social page respectively.

       <Drawer>

        <DrawerTrigger asChild className="flex justify-center items-center">

            <Button className="extra hover:opacity-80">Expanded Stats</Button>

        </DrawerTrigger>

        <DrawerContent className="p-2 flex flex-col items-center space-y-6 bg-secondary">

            {/\* Your two graphs inside the drawer \*/}

            <div className="flex justify-around w-full rounded-lg pb-4">

                <div className="grid grid-cols-2 gap-4 w-100 max-w-4xl p-2">

                    <div className="bg-white p-4 rounded-lg shadow-md max-h-25">

                        <h3 className="text-lg font-semibold">Kills</h3>

                        <p>{user.stats.kills}</p>

                    </div>

                    <div className="bg-white p-4 rounded-lg shadow-md max-h-25">

                        <h3 className="text-lg font-semibold">Assists</h3>

                        <p>{user.stats.assists}</p>

                    </div>

                    <div className="bg-white p-4 rounded-lg shadow-md max-h-25">

                        <h3 className="text-lg font-semibold">Deaths</h3>

                        <p>{user.stats.deaths}</p>

                    </div>

                    <div className="bg-white p-4 rounded-lg shadow-md max-h-25">

                        <h3 className="text-lg font-semibold">KDA</h3>

                        <p>{user.stats.kda.toFixed(2)}</p>

                    </div>

                    <div className="bg-white p-4 rounded-lg shadow-md max-h-25">

                        <h3 className="text-lg font-semibold">KAPM</h3>

                        <p>{user.stats.kapm}</p>

                    </div>

                    <div className="bg-white p-4 rounded-lg shadow-md max-h-25">

                        <h3 className="text-lg font-semibold">Wins</h3>

                        <p>{user.stats.wins}</p>

                    </div>

                    <div className="bg-white p-4 rounded-lg shadow-md max-h-25">

                        <h3 className="text-lg font-semibold">Losses</h3>

                        <p>{user.stats.losses}</p>

                    </div>

                    <div className="bg-white p-4 rounded-lg shadow-md max-h-25">

                        <h3 className="text-lg font-semibold">Win %</h3>

                        <p>{user.stats.winpercent}%</p>

                    </div>

                </div>

                <ProfileStatsBarChart last20kills={user?.stats?.last20kills || []} last20deaths={user?.stats?.last20deaths || []} last20assists={user?.stats?.last20assists || []} className="w-full"/>

                <StatsPieChart last20kills={user?.stats?.last20kills || []} last20deaths={user?.stats?.last20deaths || []} last20assists={user?.stats?.last20assists || []} className="w-full"/>

            </div>

        </DrawerContent>

        </Drawer>

The statistics drawer is triggered using a <DrawerTrigger/> component which wraps around the expand stats button, this is changed to wrap the card component and card title in the friends list and likes received list components.

Once the drawer is opened it displays the user statistics, first in a table which uses simple divs to display a grid of statistics.

export function ProfileStatsBarChart({ last20kills = [], last20deaths = [] }) {

    if (!Array.isArray(last20kills) || last20kills.length === 0) {

        return (

            <Card>

                <CardHeader>

                    <CardTitle>Match Performance</CardTitle>

                </CardHeader>

                <CardContent className="flex items-center justify-center h-40">

                    <div className="text-sm text-muted-foreground">Loading chart data...</div>

                </CardContent>

            </Card>

        );

    }

    // Build chart data from props

    const chartData = last20kills.map((kill, index) => ({

        match: `Game ${index + 1}`,

        kills: kill,

        deaths: last20deaths[index] ?? 0,

    }));

    const chartConfig = {

        kills: {

            label: "Kills",

            color: "hsl(var(--chart-1))",

        },

        deaths: {

            label: "Deaths",

            color: "hsl(var(--chart-2))",

        },

    };

Secondly the user statistics are displayed through a BarChart object imported from Shadcn which is in turn imported from Re-Charts. This bar chart pulls the user information from the populated “last20” arrays, which during the back end stats loop instead of adding to an already existing value e.g. “user.stats.kills += response.kills”, the back end appends each matches kills, deaths and assists into three “last20” arrays. This information is then imported and mapped to the chart.

            <Card className="flex flex-col" style={{ height: "450px", width: "500px" }}>

            <CardHeader className="items-center pb-0">

            <CardTitle>KDA Chart</CardTitle>

            <CardDescription>Last 20 Games</CardDescription>

            </CardHeader>

            <CardContent className="flex-1 pb-0">

            <ChartContainer

                config={chartConfig}

                className="mx-auto aspect-square max-h-[250px]"

            >

                <PieChart>

                <ChartTooltip

                    cursor={false}

                    content={<ChartTooltipContent hideLabel />}

                />

                <Pie

                    data={chartData}

                    dataKey="value"

                    nameKey="label"

                    innerRadius={60}

                    outerRadius={100}

                    strokeWidth={5}

                >

                    {chartData.map((entry, index) => (

                    <Cell key={`cell-${index}`} fill={entry.fill} />

                    ))}

                    <Label

                    content={({ viewBox }) => {

                        if (viewBox && "cx" in viewBox && "cy" in viewBox) {

                        return (

                            <text

                            x={viewBox.cx}

                            y={viewBox.cy}

                            textAnchor="middle"

                            dominantBaseline="middle"

                            >

                            <tspan

                                x={viewBox.cx}

                                y={viewBox.cy}

                                className="fill-foreground text-3xl font-bold"

                            >

                                {totalKDA.toLocaleString()}

                            </tspan>

                            <tspan

                                x={viewBox.cx}

                                y={(viewBox.cy || 0) + 24}

                                className="fill-muted-foreground"

                            >

                                KDA Ratio

                            </tspan>

                            </text>

                        )

                        }

                    }}

                    />

                </Pie>

                </PieChart>

            </ChartContainer>

            </CardContent>

            <CardFooter className="flex-col gap-2 text-sm">

            <div className="leading-none text-muted-foreground">

                Stats from your last 20 matches

            </div>

            </CardFooter>

        </Card>

Above shown is the return for the pie chart, which is displayed lastly in the drawer, which pulls all three arrays to create an informative chart, which also displays KDA Ratio (kill, death, assist ratio), which is derived from the three arrays total values’. This data is then mapped into the chart through the chartData.map function to display the chart, both of these charts are then imported and displayed within the drawer.

Both charts are also used within the user profile page which will be touched on later.

**Likes Sent & Received**

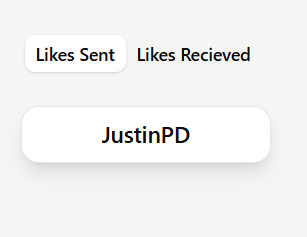


Figure 8.4.4 Home page likes sent & received tabs component.

The likes sent and received component displays the current users sent likes and received likes, these are determined based on if the user is included in a match object as liked or liker within the match.

    const fetchLikes = () => {

        local.get("/likes", {

            headers: {

                Authorization: `Bearer ${localStorage.getItem('token')}`,

            }

        })

        .then(response => {

            setLikesList(response.data);

        })

        .catch(err => {

            console.error(err);

            if (err.response?.data?.message) {

                setError(err.response.data.message);

            }

        });

    };

    useEffect(() => {

        fetchLikes();

    }, []);

    useEffect(() => {

        local.get("/liked\_me",{

            headers: {

                Authorization: `Bearer ${localStorage.getItem('token')}`,

            }

        })

        .then(response => {

            console.log("LIKES RESPONSE", response.data);

            setLikedMeList(response.data);

        })

        .catch(err => {

            console.error(err);

            if (err.response && err.response.data && err.response.data.message) {

                setError(err.response.data.message);

            }

        });

    }, []);

Likes sent and received are both called from the back-end through two get requests both called from useEffects, with the likes sent useEffect being called outside of the fetch likes function, this is to allow component updating when a new like is added.

            <Tabs defaultValue="likes" className="w-40">

            <TabsList className="flex w-full justify-around mb-4">

                <TabsTrigger value="likes">Likes Sent</TabsTrigger>

                <TabsTrigger value="liked">Likes Recieved</TabsTrigger>

            </TabsList>

            <TabsContent value="likes">

                <div className="min-h-screen">

                {likes.length === 0 ? (

                    <p className="ml-2">No likes sent yet!</p>

                ) : (

                    <div className="grid grid-cols-1 gap-4">

                    {likes.map((like) => (

                        <MatchCard key={like.liked\_id} user={like.liked} />

                    ))}

                    </div>

                )}

                </div>

            </TabsContent>

Both of these user lists are then displayed under two tabs and each with their own cards, the likes sent card is very simple and just includes a title.

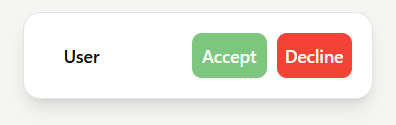


Figure 8.4.5 Likes R+teceived card.

On the other hand the likes received card has more functionality with two Shadcn buttons being included each with their own function.

The accept button calls the back ends match accept route andthen uses a useNavigate function to route to the social page. The decline button calls the match decline function in the back end and reloads the homepage.

**Starred Users**

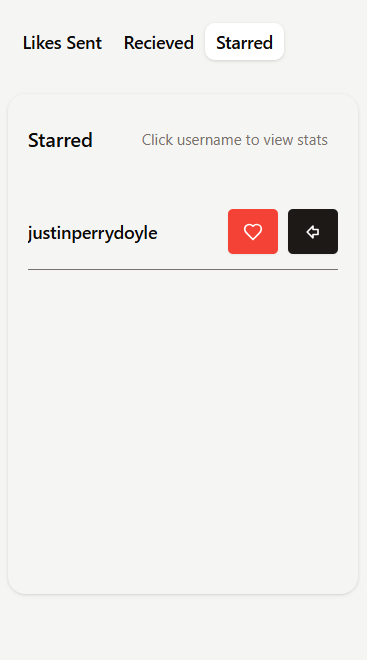


Figure 8.4.6 Starred users list.

The starred users list is a display of users that have been starred by the currently active user, two buttons are displayed on the card, to like and undo a star for the user. Each of these cards call their respective “like user” and “decline user” routes.

                            <ScrollArea className="h-[320px] px-4 py-2">

                                {starred.length === 0 ? (

                                    <p className="text-sm text-muted-foreground">No starred yet!</p>

                                ) : (

                                    starred.map(starMatch => {

                                        const friend = starMatch.liked

                                        console.log("match", friend);

                                        return (

                                                    <Drawer>

                                                    <button key={friend.id} className="flex justify-between items-center py-3 border-b border-muted-foreground w-full">

                                                    <DrawerTrigger asChild className="">

                                                        <span className="text-sm font-medium">{friend.username}</span>

                                                        </DrawerTrigger>

                                                        <div className="flex items-center space-x-2">

                                                        <Button onClick={() => onPress(starMatch.id)} className="danger outline hover:opacity-80 text-white font-bold py-2 px-4 rounded w-auto">

                                                            <HeartIcon className="w-4 h-4" />

                                                        </Button>

                                                        <Button onClick={() => onDecline(starMatch.id)} className="hover:opacity-80 text-white font-bold py-2 px-4 rounded w-auto">

                                                            <ArrowBigLeft className="w-4 h-4" />

                                                        </Button>

                                                        </div>

                                                    </button>

The starred cards are held in a scroll area component from ShadCN and open a drawer component the same as the user card.

## Social Page

The social page is where all of the users friends and groups are managed, this is the social hub of FindMyTeam and displays users groups and friends list to them. The social page also contains updating notifications and new tags on each unseen group.

**Groups**

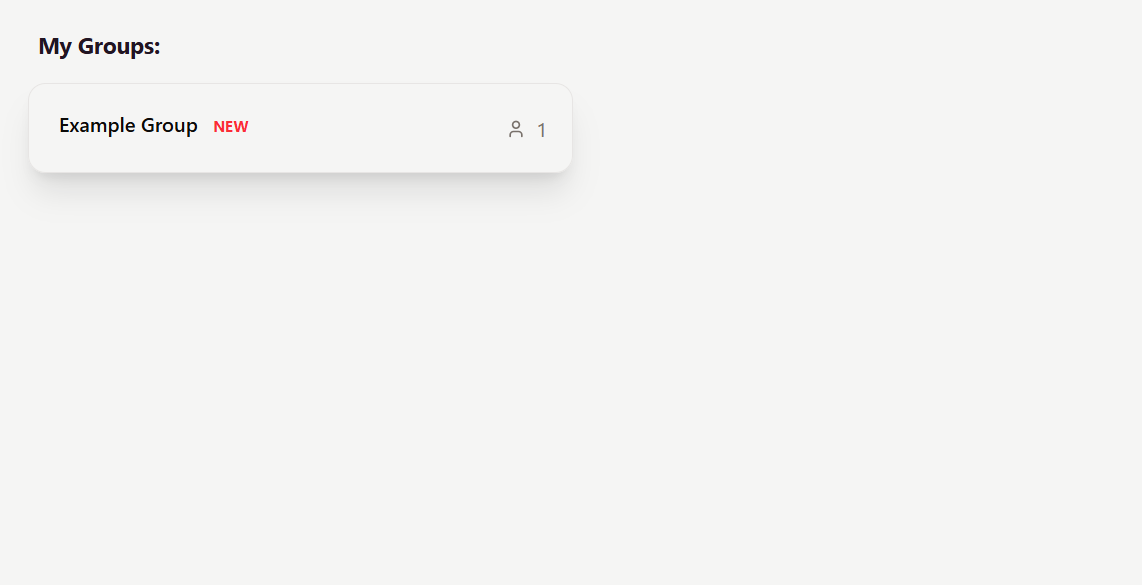


Figure 8.5.1 Groups card being displayed on the social page.

Groups are displayed on the social page in a grid of cards, the group cards are relatively simple, showing the group name, number of members and a new tag within the Shadcn card.

                <Card className="w-full h-18 bg-base-100 shadow-xl" onClick={handleClick}>

                    <div className='flex items-center justify-between'>

                    <CardHeader className="w-full">

                        <CardTitle><button>{group.group\_name}</button>   {!group.seen && (

                        <span className="ml-2 text-xs text-red-500 font-bold">NEW</span>

                    )}</CardTitle>

                    </CardHeader>

                    <div className="flex items-center space-x-2 mr-5">

                        <User className="w-4 h-4 text-muted-foreground" />

                        <span className="text-muted-foreground">{group.users ? group.users.length : 0}</span>

                    </div>

                    </div>

                </Card>

The social page calls the groups using the “/groups” get request on the back end, and feeds that information to the group cards as a prop and through a map function. Above seen is the layout of the card once the information has been passed to it.

function GroupNotifications() {

    const [hasNewGroup, setHasNewGroup] = useState(false);

    const [local] = axios;

    const checkForNewGroup = () => {

        local.get('/notifications/groups', {

            headers: {

                Authorization: `Bearer ${localStorage.getItem('token')}`,

            },

        }).then(res => {

            setHasNewGroup(res.data.length > 0);

        }).catch(err => {

            console.error("Polling error", err);

        });

    }

    useEffect(() => {

        checkForNewGroup(); // Initial check

        const interval = setInterval(() => {

            checkForNewGroup(); // Check every 5 seconds

        }, 5000);

        return () => clearInterval(interval); // Cleanup on unmount

    },[]);

    return hasNewGroup;

}

export default GroupNotifications;

To allow groups to have a new group tag and to allow red dot notifications on the FMT sidebar a group notifications function was added to the front end, this function uses polling to ping the back end every 5 seconds with a “hasNewGroup” check. If a new group is shown the sidebar is updated with a red dot above the social navigation button.



Figure 8.5.2 The sidebar component with a notification dot above social.

To allow for a group to have a “new” tag each group has a Boolean value called seen, this value is updated for each individual user in the group through the UserGroup table, allowing the seen value to be checked and new tag to be displayed if the user has not previously displayed the group.

**Friends List**

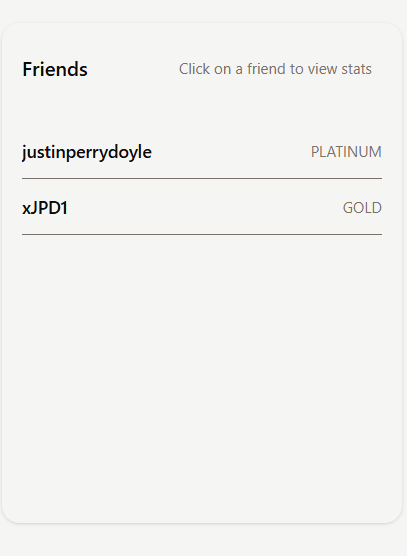


Figure 8.5.3 User friends list on social page

When a user like request is accepted they are routed to the social page, this will display a new friend in their friends list component as shown above. The friends list is a scroll area that displays simple user information such as rank and username, each users username can be clicked on which will then display their respective stats drawer.

    useEffect(() => {

        local.get("/friends", {

            headers: {

                Authorization: `Bearer ${localStorage.getItem('token')}`,

            }

        })

        .then(response => {

            console.log("RESPONSE", response.data);

            setFriendsList(response.data);

        })

        .catch(err => {

            console.error(err);

            if (err.response && err.response.data && err.response.data.message) {

                setError(err.response.data.message);

            }

        });

    }, [local]);

Friends are called using a standard useEffect.

                            <ScrollArea className="h-[320px] px-4 py-2">

                                {friends.length === 0 ? (

                                    <p className="text-sm text-muted-foreground">No friends yet!</p>

                                ) : (

                                    friends.map(friendMatch => {

                                        const { liker, liked, current\_user\_id } = friendMatch;

                                        const friend = liker.id === current\_user\_id ? liked : liker;

                                        return (

                                                    <Drawer>

                                                    <DrawerTrigger asChild className="">

                                                    <button key={friend.id} className="flex justify-between items-center py-3 border-b border-muted-foreground w-full">

                                                        <span className="text-sm font-medium">{friend.username}</span>

                                                        <span className="text-xs text-muted-foreground">{friend.rank}</span>

                                                    </button>

                                                    </DrawerTrigger>

Each friend has their information displayed, which also acts as a drawer trigger. (Drawer code continues below code snippet and is identical to the user drawer code from home page)

**Create Group**

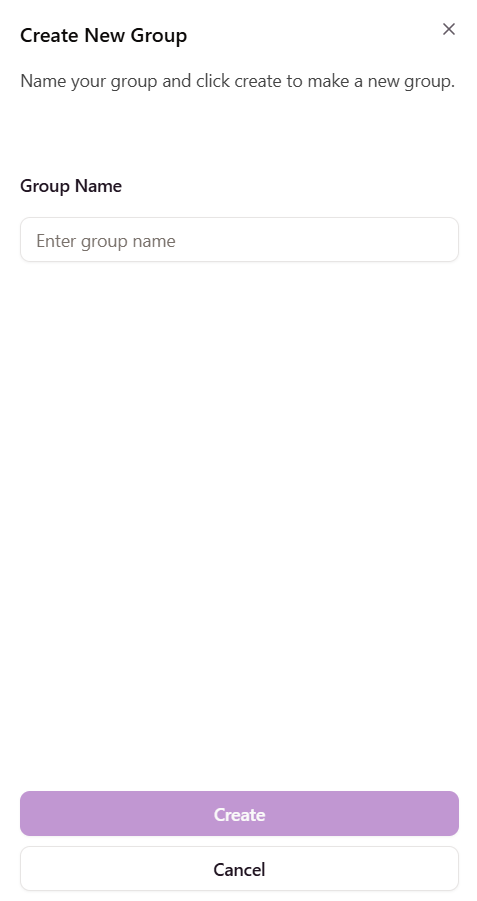


Figure 8.5.4 Group create component.

The create group component is a simple form field post component with one input being the group name. It uses a Shadcn Sheet component with Inputs and Buttons also utilised.

**Single Group Page**

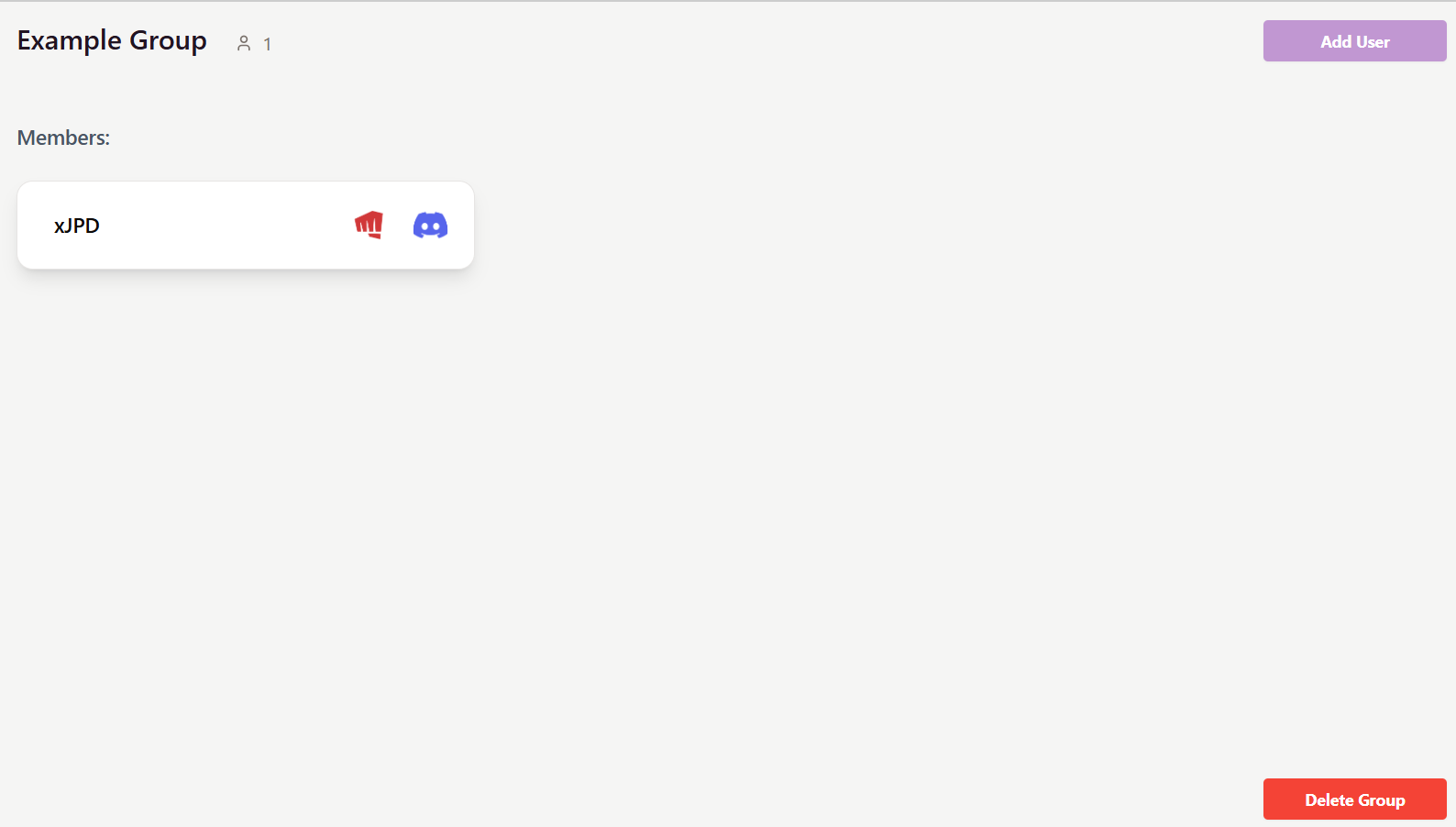


Figure 8.5.5 Single group display page with one member.

The single group page is used to display user information to other users, groups are created to allow players to connect with each other, here users can create a team of players, all of which can see each other’s discord and riot information with the click of a button, allowing quick, seamless transitions to discord for communication purposes, and to League of Legends itself, where they can copy and paste their displayed RIOT information once the riot logo (which is a Shadcn Popover component) is clicked on. The discord logo is clickable and is a direct link to that users discord account which, once clicked, users will be redirected to.

const GroupPage = () => {

            const { group\_id } = useParams();

            const [error, setError] = useState(null);

            const [group, setGroup] = useState([]);

            const [local] = axios;

            const [users, setUserList] = useState([]);

            const navigate = useNavigate();

            const [loading, setLoading] = useState(true);

            useEffect(() => {

                local.get(`/groups/${group\_id}`,{

                    headers: {

                        Authorization: `Bearer ${localStorage.getItem('token')}`,

                    }

                })

                .then(response => {

                    console.log("RESPONSE", response.data);

                    setGroup(response.data);

                    setUserList(response.data.users);

                    setLoading(false);

                })

                .catch(err => {

                    console.error(err);

                    if (err.response && err.response.data && err.response.data.message) {

                        setError(err.response.data.message);

                    }

                });

            }, [local]);

On the group page the group is called through the URL parameters passed during the navigation from the main social page, the GroupPage then queries the back end upon page mount using a useEffect using the “/groups/${group\_id}” URL. Upon data return, the group, list of users in the group (sent with the group through the association table and to\_dict() functions in the back end) and loading state are set. These are then all returned using ShadCN cards with the user information displayed within each.

        <Card className="w-100 shadow-lg p-2">

        <div className='flex justify-between items-center'>

        <CardTitle className="text-lg font-semibold mx-6">{user.username}</CardTitle>

        {/\* Card Header: Username, Riot Logo (Popover), and Discord Logo \*/}

        <div className="flex items-center justify-around">

        <div className="flex items-center space-x-4">

            {/\* Riot Logo with Popover \*/}

            <Popover>

            <PopoverTrigger asChild>

                <img src={riotLogo} alt="Riot Games Logo" className="w-12 h-auto cursor-pointer hover:opacity-80 transition" />

            </PopoverTrigger>

            <PopoverContent className="text-center w-30 h-15">

                <div className="flex flex-col items-center space-y-1">

                <p className="font-semibold">{user.riot\_name} #{user.riot\_tag}</p>

                </div>

            </PopoverContent>

            </Popover>

        </div>

        {/\* Discord Logo \*/}

        <CardDescription className="text-muted-foreground">

            <a

            href={`https://discord.com/users/${user.discord\_id}`}

            target="\_blank"

            rel="noopener noreferrer"

            className="hover:opacity-80 transition"

            >

            <img src={discordLogo} alt="Discord Logo" className="w-15 h-15" />

            </a>

        </CardDescription>

        </div>

        </div>

        </Card>

The code for the user card displaying the RIOT and Discord logos, both clickable buttons to help users connect seamlessly.

**Add User**

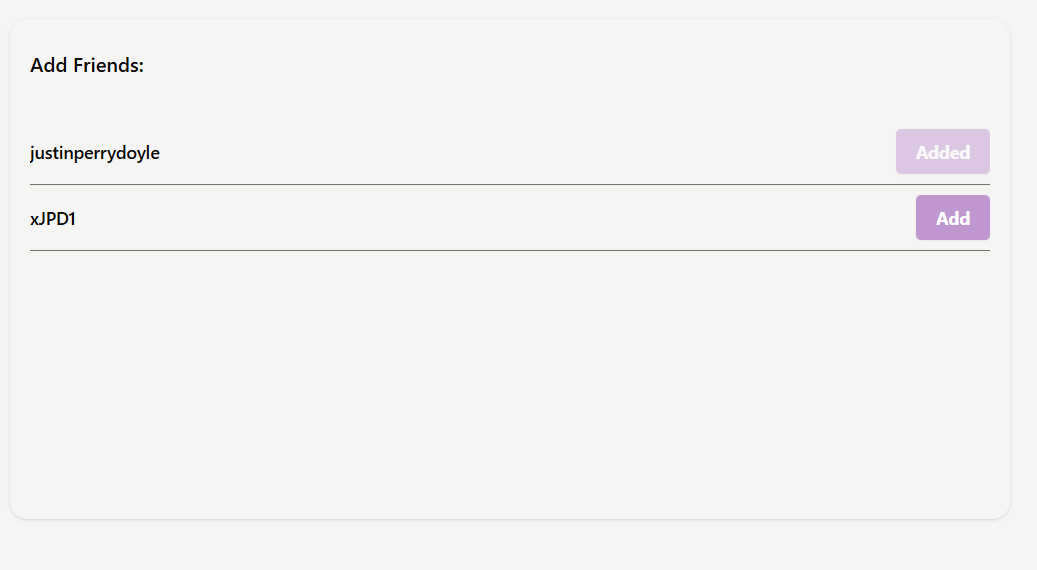


Figure 8.5.6 Add users component.

The add users component uses a scroll list similar to that of the friends list on the main social page, the difference between the two is that the add friends component displays an add user button for each user in the list.

                <div className="flex justify-center items-center min-h-screen p-4 w-full bg-secondary">

                <Card className="bg-muted h-[400px] border-none w-200">

                        <CardHeader className="flex items-center justify-between pl-4">

                            <CardTitle className="text-md">Add Friends:</CardTitle>

                        </CardHeader>

                        <CardContent className="p-0">

                            <ScrollArea className="h-[320px] px-4 py-2">

                                {users.length === 0 ? (

                                    <p className="text-sm text-muted-foreground">No friends yet!</p>

                                ) : (

                                    users.map(friendMatch => {

                                        const { liker, liked, current\_user\_id } = friendMatch;

                                        const friend = liker.id === current\_user\_id ? liked : liker;

                                        return (

                                            <div key={friend.id} className="flex justify-between items-center py-2 border-b border-muted-foreground">

                                                <span className="text-sm font-medium">{friend.username}</span>

                                                <Button className="primary hover:opacity-80 transition text-white font-bold py-2 px-4 rounded" onClick={handleClick(friend.id)} disabled={isUserInGroup(friend.id)}>{isUserInGroup(friend.id) ? 'Added' : 'Add'}</Button>

                                            </div>

                                        );

                                    })

                                )}

                            </ScrollArea>

                        </CardContent>

                    </Card>

Within the user list (which is called the same as the friends list using the “/friends” query from a useEffect) each user card is displayed, and a button for each, the users.id is then checked against the id’s of the group members to attain if that user is already a member of the group or not, if so the add user button is changed to disabled and “added” is shown instead of “add”.

        const isUserInGroup = (userId) => {

            return group.users.some(user => user.id === userId); // Check if user is already in the group

        };

The query above checks if the user is present in the group.

        const handleClick = (id) => () => {

            local.post(`/add\_user\_to\_group`,{

                group\_id: group\_id,

                user\_id: id

            },{

                headers: {

                    Authorization: `Bearer ${localStorage.getItem('token')}`,

                }

            })

            .then(response => {

                console.log("RESPONSE", response.data);

                navigate(`/groups/${group\_id}`)

            })

            .catch(err => {

                console.error(err);

                if (err.response && err.response.data && err.response.data.message) {

                    setError(err.response.data.message);

                }

            });

        }

The add user function is called upon press of the add button, the users id is stored and sent to the back end along with the group id, this allows a connection to be made and the user is successfully added to the group.

**Delete Group**

        const handleDelete = () => {

            local.put(`/groups/${group.id}/delete`,{},{

                headers: {

                    Authorization: `Bearer ${localStorage.getItem('token')}`,

                }

            })

            .then(response => {

                console.log("RESPONSE", response);

                navigate(`/social`)

            })

            .catch(err => {

                console.error(err);

                if (err.response && err.response.data && err.response.data.message) {

                    setError(err.response.data.message);

                }

            });

        }

The delete group function simply calls the back end using the user id within the URL to produce a soft delete for the group, this means the groups hidden attribute will be changed to true, hiding the group from all users involved.

## Profile Display

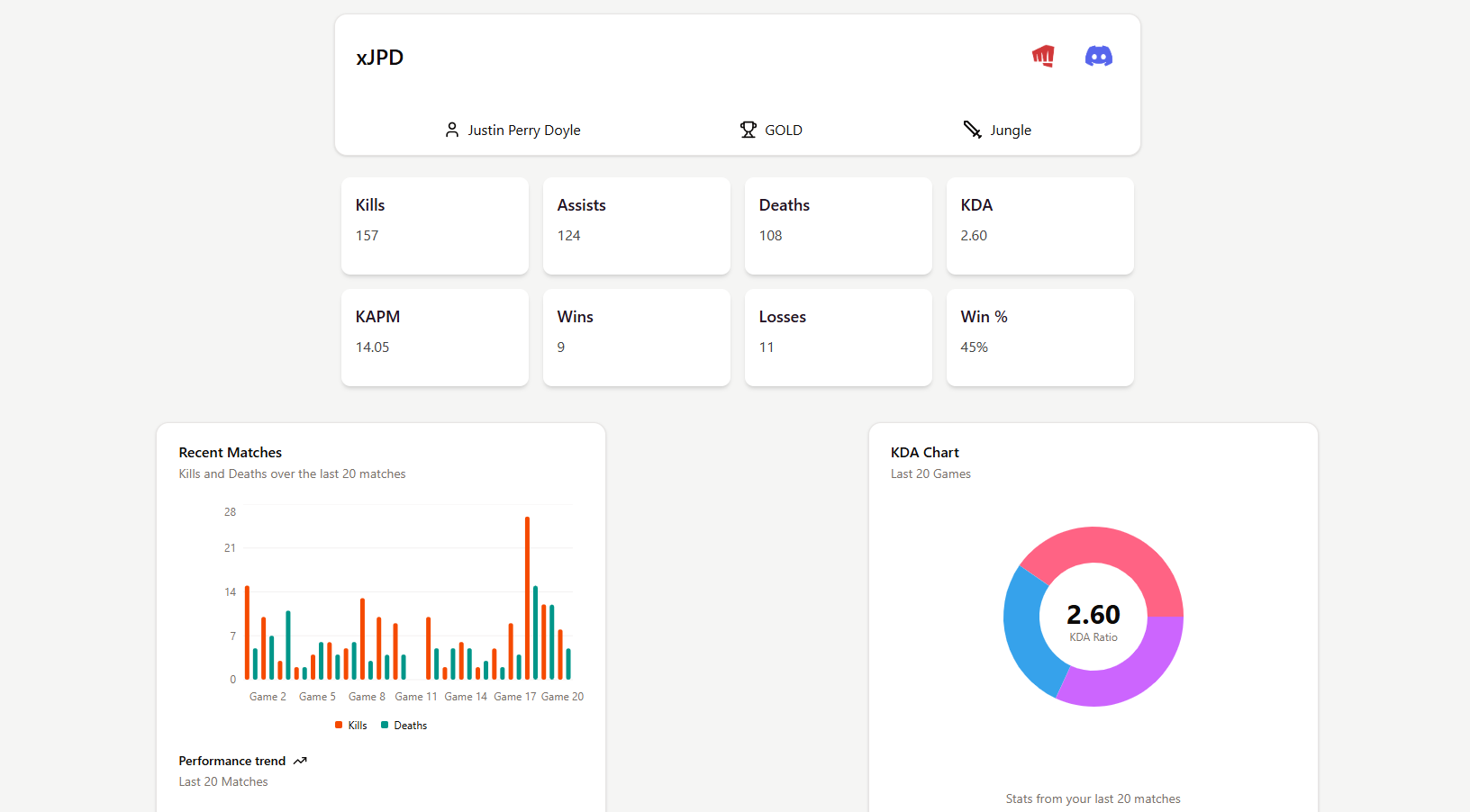


Figure 8.6.1 Profile page for FindMyTeam.

The profile display for FMT is largely the same information as the user card information along with the drawer used to display expanded stats for each user. This has been put into one page so as to allow each user to see their own account in a single page along with their statistics and information.

The code for the user single profile page is almost exactly identical to the code for the user card and drawer.

## Helper Page

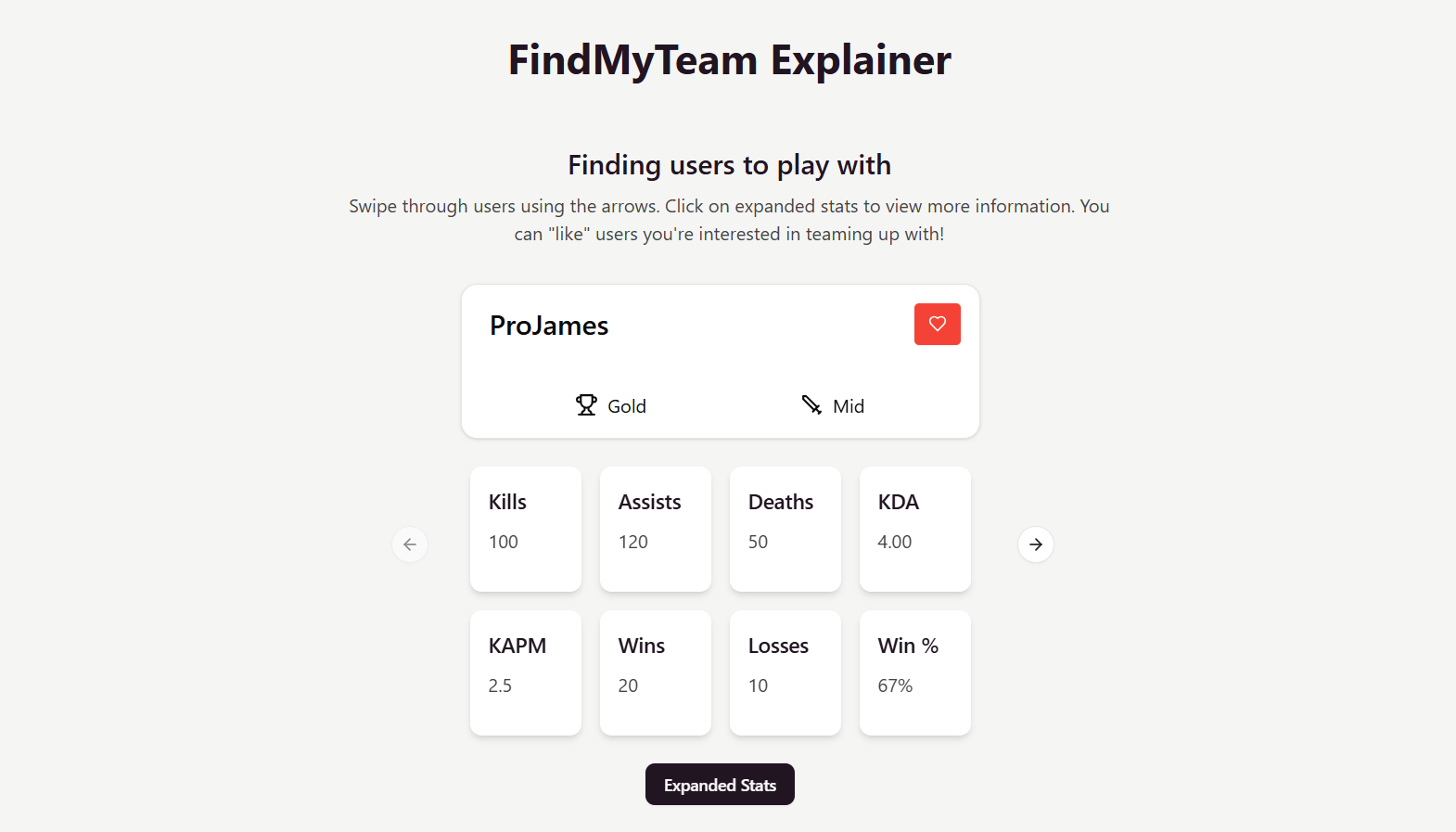


Figure 8.7.1 The top of the FindMyTeam explainer/helper page.

The FMT helper page is designed to allow users who are unfamiliar with FindMyTeams systems, or simply need a refresher when loading the application up again after a break. For users who may struggle with understanding technology and utilising applications but want to avail of the service FMT provides the helper page allows those users to explore the application without having to worry about the consequences of clicking a wrong button leading to irreversible actions.

The helper page displays multiple components from FindMyTeam to the user with instructions shown for each, in the above figure it shows the interactive user card component with placeholder user information included and a worded explainer above describing the cards functionality and how to use. There are multiple cards filled with fake user data and stats for a user to look through on this specific component.

        <Card className="w-70 shadow-lg p-4">

            <div className="flex justify-between items-center">

            <CardTitle className="text-sm ml-4">{fakeUser.username}</CardTitle>

            <div className="flex items-center space-x-2">

                <Button className="w-15 happy hover:opacity-80">Accept</Button>

                <Button className="w-15 danger hover:opacity-80">Decline</Button>

            </div>

            </div>

        </Card>

The code shown above here is a practice user like received card, it shows and with its explainer text explains to the user how likes work, how to accept and decline them and what will happen once those choices are made.

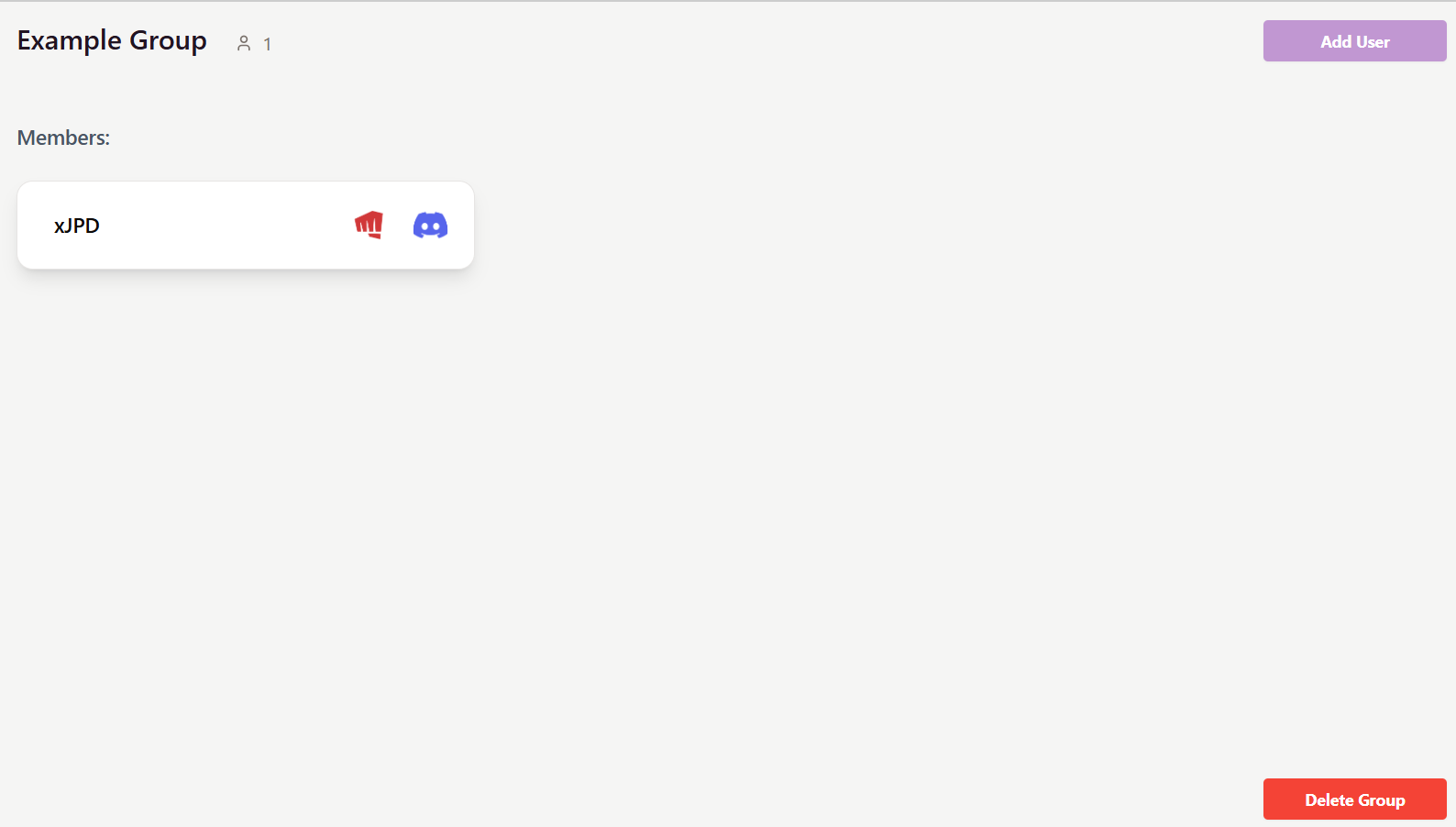


Figure 8.7.2 A user card which would be displayed in the group page.

Lastly explained and displayed is a user card that would be displayed inside of a group, it is explained that this is how users connect, and that each buttons are clickable and perform different actions, the card is interactive and allows the user to get a feel for the components before actually entering the app.

# Testing

## Introduction

Testing is an integral part of the creation of any application and was carried out throughout FindMyTeams development. Two types of development were carried out:

**Functional Testing**

Carried out all throughout the development of FindMyTeam. From the first user model created in the back end up until the final component completed in the front end functional testing was being completed alongside to verify and streamline development.

**User Testing**

Carried out towards the end of and when FMT’s development was complete, user testing was used to inform the places in which FMT may struggle or be lacking.

## Functional Testing

Most of the functional testing for FindMyTeam was white and black box testing. White box testing is a form of testing where the tester has access to the code of the application and full knowledge of its functionality and design documents. Black box testing on the other hand is concerned with specific unit instances and their responses.

Both forms of testing were mainly done on Insomnia Client which is an API client service designed to allow developers to test and run back end code without the need for a functioning front end application to ingest the data.

Insomnia allows a user to make a call to a back end and displays the response in a dedicated section within the applications environment. This allows for code to be tested quickly and efficiently, where the tester is aware of the expected outcomes during the black box phase, and then tests for edge cases after moving to white box testing.



Figure 9.2.1 Insomnia client displaying an expected outcome for the get users route.

### CRUD + Auth

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test No | Description of test case | Input | Expected Output | Actual Output | Comment |
| 1 | User Create/Register | Form object containing user information | Status 201 message + string success message | Status 201 code and message | User create worked as expected. |
| 2 | Users Get | Empty request with no headers. | Status 401 unauthorized + error message | Status 401  Code + message | Users get auth working. |
| 3 | Login | Form containing previously created user information | Status 200 + access code + message | Status 200 + access code + message | Login working as expected. |
| 4 | User Update | Form field containing user information to be updated | Status 201 + success message | Status 201 + success message | User update works accordingly. |

### API Ingestion & Connection

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test No | Description of test case | Input | Expected Output | Actual Output | Comment |
| 1 | Discord Login/Callback allowing the user to attach their discord id to their account. | Discord login route call from back end with token attached | Status 200 + Status 201 (when discord is updated on second call) + success message + user discord\_id update | Status 200 + status 201 + success message + discord id updated | Took many tries to achieve desired output, may need to be changed during front end development. |
| 2 | RIOT API stats ingestion during the registration of the user account, placement into stats object and connection to user. | User register information RIOT information included | Status 201 + success message + stats object creation and user object return. | Status 201 + success message + stats object created + user object return | Complicated process, trial and error was needed until this test was fully working. |

### Groups & Matches

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test No | Description of test case | Input | Expected Output | Actual Output | Comment |
| 1 | Group creation and addition of creator user id into said group. | Form information with group information, JWT token headers for user auth and identification. | Status 201 + success message + group creation inside of database. | Status 201 + success message + group creation inside of database. | Group creation worked well, JWT identify worked well with the user token. |
| 2 | Request match with user who the current user has already matched with. | Post request including the liked user id in the URL parameters + JWT token for current user identification. | Status 409 +  Error conflict message | Status 409 + error message | Matching with users who the current user has already matched with is not possible now. |
| 3 | Group get query | Get request to attain the information for a given group. | Status 200 + success message + group return including connected users data | Status 200 + success message + group return including connected users data | Group and users data is returned successfully at this point. |

### Discussion of Functional Testing Results

The tests shown above are only a fraction of the total number of tests completed through insomnia. Hundreds of individual tests were completed on the routes before the final functional tests were completed.

These included tests for as many edge cases as possible, and each of these tests would go on to inform how the front end of FMT would deal with these edge cases and errors.

FindMyTeams’ front end has to prioritize user experience, limiting the number of mistakes a user can make on a given page and eliminating catastrophic errors that could completely derail the site or user flow.

The functional testing results were satisfactory in every case after rigorous testing was completed on every route throughout development.

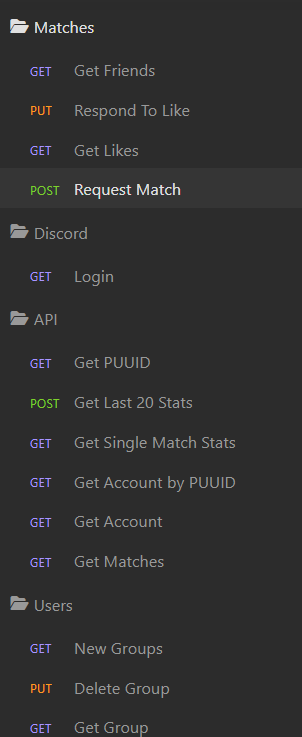


Figure 9.2.4.1 Some of the many routes used in Insomnia client during development and functional testing.

## User Testing

Two user tests were completed towards the end of development for FindMyTeam, these tests were conducted in a controlled environment, where only the tester and participant were present, no communication was given by the tester to the participant during the test and only the goals and necessary information were provided to the participants before the tests began. User feedback was collected after each test and that information was invaluable in making changes to FMT that were overlooked previously.

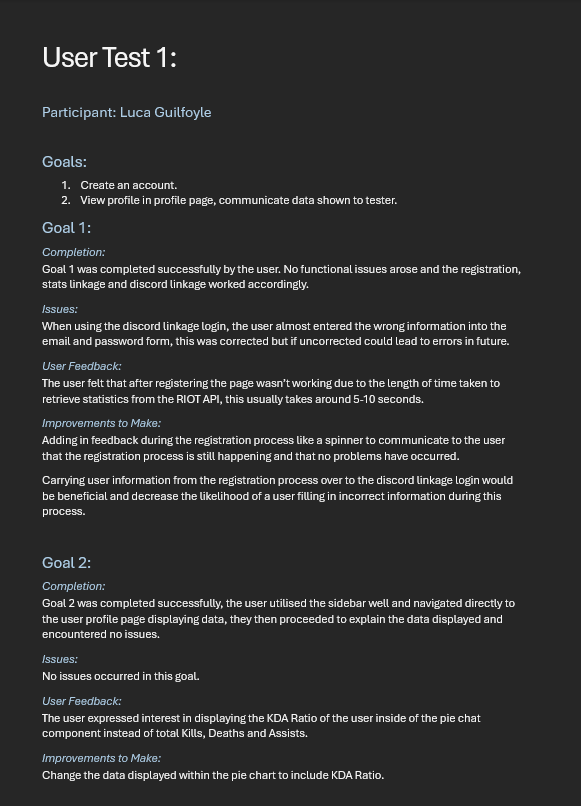


Figure 9.3.1 First user test for FMT.

In the first user test the register and login functionality was tested, along with the data communication and display through the profile page. The user responded positively to the application and had some valuable feedback after the test was completed. Improvements were made to the application based on the feedback and issues that arose.

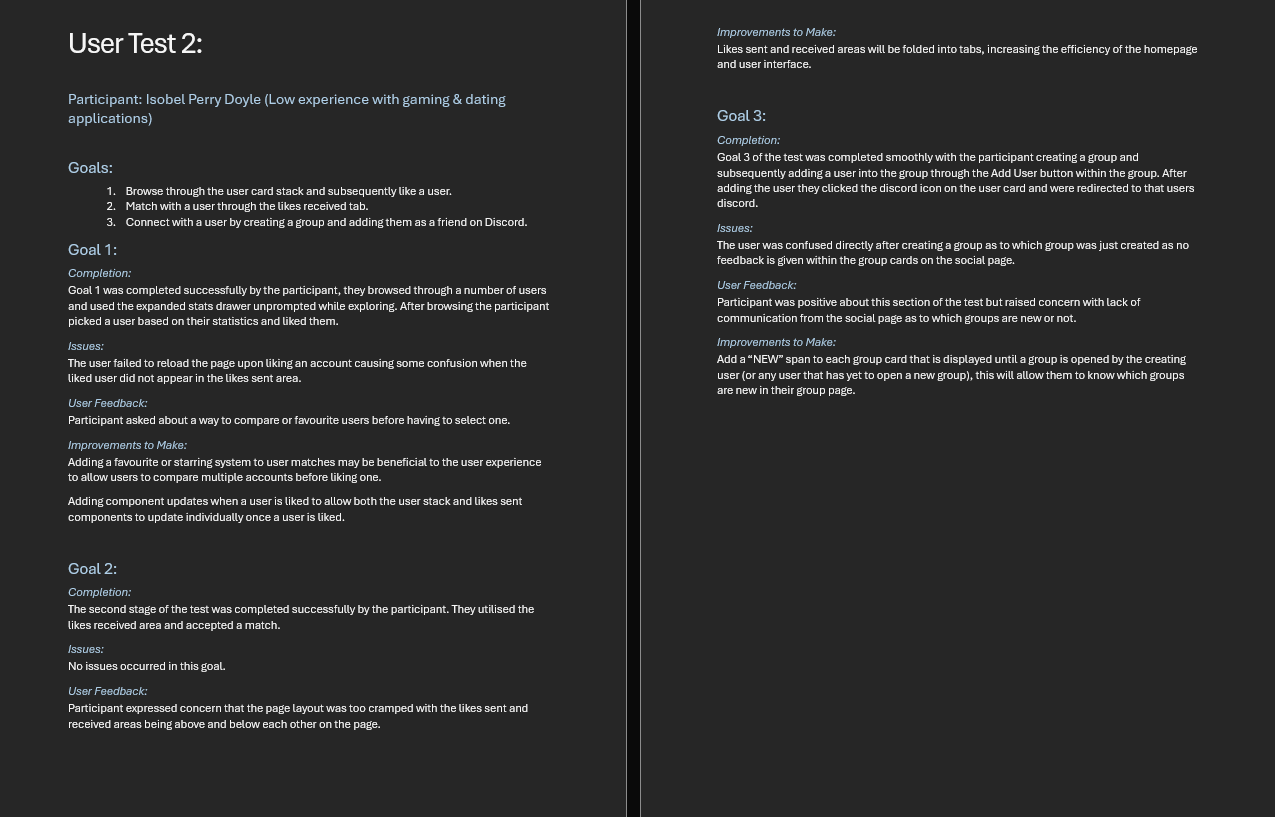


Figure 9.3.2 Second test completed for FMT.

The second test for FMT was positive and gave important feedback on multiple systems within the application to improve on which were subsequently completed and implemented into FindMyTeam.

The second participant was a less experienced user with apps similar to FMT, this was important as to gain a full understanding of all types of users from experienced to brand new gamers, both need to be included in the vision, development and testing for FMT.

## Conclusion

Testing was an important part of the development of FMT and allowed the identification of multiple areas where development had lacked or was not functioning as intended for the application. This section helped to diagnose and treat these underlying problems to make FMT the best application it can be.

Functional testing was completed throughout the development and dealt with the handling of back end data and error response as well as front end functionality. This allowed these development areas to be handled quickly and efficiently meaning no major rewrites or site breaking errors were allowed to fall through the cracks.

User testing was invaluable in identifying where development had lacked both in functionality and in user experience, done towards the tail end of development this allowed for some important improvements to be made to the application meaning the user experience is as seamless as possible.

# Project Management

## Introduction

Project Management was abided by from start to finish of the development of FMT. Research around the market for such an application, then how that application would be developed were done to gather a solid base and create a plan for the next stages of development.

Once development began the SCRUM methodology was utilised with sprints being held every week, this allowed team communication and guidance at regular intervals, planning for each individual sprint and important debriefing and issue handling within the development of the application.

Project management was also aided by the inclusion of different external systems and tools. These tools include Miro, an online software where teams can share and collaborate on work, which was used to put together sprint collages with information on goals, problems and plans. These collages also allowed screenshots of work completed to be viewed allowing for supervisor viewing of progress even outside of meeting times.

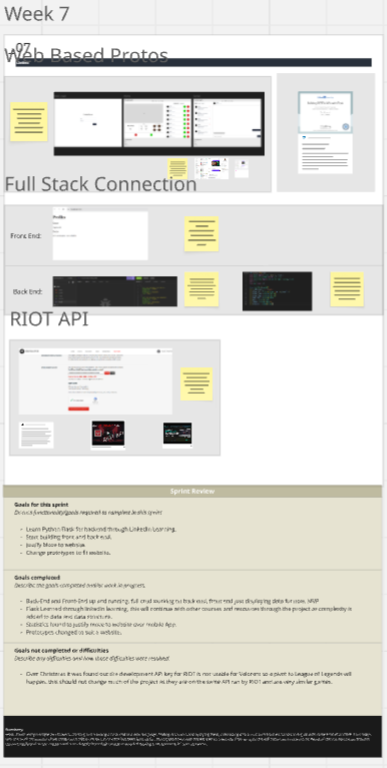


Figure 10.1.1 Example of a sprint week displayed in Miro.

Along with Miro, GitHub was utilised for version control and later to host the application through Render (back end & database) & Vercel (front end). Microsoft Teams was then used to facilitate meetings between developer and supervisor and to allow for a consistent communication channel through the messages section.

## Project Phases

### Proposal

The project proposal phase was used to identify the problem to be solved by FindMyTeam and how this application would be utilised by users to solve said problem. A project proposal was sent to supervisors and lecturers to discuss this topic and to verify that the problem itself was valid, and the development of the application would be possible.

Included in the project proposal were the details of FindMyTeams purpose, what tools, languages and systems would be used in the development of the application, the possible issues that may arise during development and the minimum viable product that would be expected at the end of the development cycle. Also included was a breakdown of the project management involving how and how often the developer would meet with their supervisor and how these meetings would be held to maximise the efficiency and effectiveness of the project.

An issue found during the proposal phase was that using a recommender system within FMT would not be attainable. This was due to the issues around cold start and recommender systems. Gathering a solid base of data to use in the training of a model was deemed not possible in relation to the application. FindMyTeam then pivoted towards using systems such as filtering and randomising the user stack to create engaging user encounters on the application and giving these users control over who they see on FMT. This information would not have been possible to attain without the proposal and previous research phase on neural networks and recommender systems.

### Requirements

The requirements phase of the project dealt with finding and confirming the functional and non-functional requirements for the development and deployment of FMT. Nothing unexpected was found during this phase of development. Functionally FMT would use many different technologies such as, React-Vite, PyFlask, SQLLite, PostgreSQL and others. Non-functionally FMT would need to be secure, efficient and be appealing to look at and be utilised by users.

To figure out said requirements for FindMyTeam interviews were held with users, these interviews helped to determine what users were looking for when asked about an application like FMT. Also researched were similar applications like Tracker Network - a data driven application designed to allow users to connect their RIOT accounts and view their statistics, and Tinder – a dating application which uses a swipe gamification feature to match users together into romantic and platonic pairings. These applications would show how FMT should behave, and what gaps it would have to fill, which these applications lacked or fell short on.

Also completed during this phase of development were personas, empathy and journey maps, this brought many things to attention including the need for a helper page and more inclusion of new and inexperienced users.

### Design

The design phase of project management is held to ensure that FMT looks and functions as expected once it is completed. In this phase the functional design, including how the application would look, through wireframes, font maps and a colour palette to decide the UI design of FMT.

For the functional design architecture diagrams for both the front and back end were completed along with file structure diagrams for both. The database was designed using an ERD diagram and deeper database schema detailing all relationships and tables to be utilised within FMT. This allowed the development of FMT to be streamlined and seamless.

Issues and improvements that arose during this section included folding the user and profile tables into a single table under the user title instead of splitting those tables up.

### Implementation

Implementation was the largest section of the project phase, during the implementation phase the back end and front end for FMT were built, the database was created, changed (PostgreSQL) and hosted, and the application was designed with the use of Shadcn and Tailwind.

During implementation FMT faced many challenges and hurdles, but all were worked through in time, nothing major was changed during the implementation phase of development, although within the application each day compromises were made and systems were changed to allow them to be created and function properly within the context of FMT. Examples would include the move away from sessions in the back end to define a currently active user, and towards decrypting JWT tokens as this was more secure and efficient.

The implementation phase is the most important phase of any applications development and was handled well with great communication and support between all parties involved in it, all facilitated by the previous work done in the proposal, requirements and design stages of the project.

### Testing

User testing was the final stage of the project, this was held at the end of the project development and was held to give insight into where FMT may have fallen short from the point of view of users. Tests were conducted with two participants, one with high experience and one with low experience, this was to give a varied view of the applications approachability to new users. Both users gave valuable feedback and issues were found and treated during this phase of development.

The functional testing phase was done throughout the application and consisted of consistent unit testing in Insomnia client. This testing was done consistently for every function and route on the back end of FMT, each edge case was tested for issues, proper responses, success + error messages and to make sure that error handling was being handled correctly.

## SCRUM Methodology

A diagram of a scrum process

AI-generated content may be incorrect.

Figure 10.3.1 Diagram of the SCRUM methodology (AndreaB, 2021).

FindMyTeam used the SCRUM methodology in its implementation. SCRUM works on the principle of sprints. Each sprint can be between 1-4 weeks long depending on the project in question, each sprint will include a backlog of work to be completed, a plan usually ideated in the preceding sprints conclusive meeting and then an implementation phase.

For FMT weekly sprints were used from December 2024 through to the start of May 2025. Each sprint consisted of a backlog, usually carried over from previous sprints, a plan of action, ideated during the conclusive meeting of previous sprint, and the sprint itself where the plan was implemented into production. This would all be followed by a conclusive meeting in which progress would be discussed and the plan for the next sprint would be thought up.

The application had a backlog of work to be done, each sprint meeting work would be pulled from the backlog to be done during the sprint, any uncompleted work would be put back onto the backlog or more likely, transferred over to the next sprint to be completed. Work was pulled from the backlog based on its urgency and requirement for other functionality, e.g. create user would need to be implemented before display all users.

In FMT’s case the SCRUM team only consisted of two people, being the project supervisor and developer. The project supervisor was in essence the SCRUM master and the developer was consistent with the SCRUM team, with there being no specific product owner the project was led as a team effort to decide what products and functionality needed to be included.

## Project Management Tools

### GitHub

GitHub was used for the version control of FMT. Each day or multiple times a day whenever code was added, updated or changed a “git add”, “git commit” and “git push” command were used to push the code to a linked GitHub repository. This would increase the redundancy for the application both front and back end, meaning if a serious issue occurred either in development or in the hardware/hardware storage of a device that a rollback to a previous recent version of FindMyTeam could be completed.

GitHub was also used for both the front and back end of the application for hosting, as both Render & Vercel use the application GitHub repo link to deploy code to their servers.

GitHub as usual was easy to work with and seamless in the development of FMT.

### Miro

Miro was utilised in the project management of FMT, where collages were made each week for each sprint, in these collages important information including goals, issues, and completed tasks were shown along with screenshots and important contextual information for each completed task.

Miro was incredibly helpful and made project management much easier.

### Teams

Microsoft Teams was used as the main communication platform for members of the development team, the calls feature was used for weekly meetings and the ability to share screens displaying information and problems to other users was very helpful during each sprint. Also used as the main non-vocal communication channel Teams’ messaging feature was used as a consistent text channel for users within the team in case of more urgent issues or meeting planning.

## Reflection

### Your views on the project

FindMyTeam was a very enjoyable project to work on, from my perspective I got to learn a lot of new things, Python for instance was something I had wanted to work with for a long time and this project allowed me to do that. I feel the project went very well, the development was smooth, the work was spread out and planned well, meaning cramming periods weren’t required, although of course approaching project submission more time was spent coding and writing. There were no large issues with the project through its development and once completed almost all of the MVP was fulfilled in the final application including some extra functionality that were added on top. In this regard I’m incredibly happy with how FindMyTeam turned out and performs in the hands of users.

### Completing a large software development project

Completing a project as large as FindMyTeam put a lot of pressure on pre-planning and being disciplined throughout the projects development process. Getting work done early on in the development cycle was incredibly important and was facilitated by consistent communication between developer and supervisor. This constant guidance allowed for me to stay on top of my work, in some ways due to the accountability of having sprint meetings each week. This project allowed me to learn new languages, like Python which I had not used before in any kind of full stack or back end capacity. Working with Shadcn was enjoyable too, although new their docs and systems make it an incredibly easy library to utilise. SQLLite and PostgreSQL are also new to me but neither were particularly deep or difficult to learn as they are much more hands off.

Due to the size of this project it has taught me how to stay on top of more work at one time, at preplanning and having a vision for each individual part of the process.

### Working with a supervisor

Working with a supervisor I felt was one of the best parts of the project, it allowed me to be held accountable by someone in a position to do so in a guided and positive manner, I felt my supervisor was always communicative and interested in my project and giving me the best tools possible to allow for my success at all stages. We held weekly meetings almost every week and they had complete trust in my ability to do the work and create the project I had envisioned.

### Technical skills

Technically I have learned a multitude of different skills, Python as a language in the context of building a REST API and Flask as a framework to do so. SQLLite & PostgreSQL are both valuable skills to have learned in the process of making FMT as well. This was also my first time working with Shadcn which is a brilliant framework/library for design components.

### Further competencies and skills

Being able to communicate with a supervisor throughout the project was invaluable and a massive learning experience for me, this was the first time I have had regular meetings and an almost professional setting while developing a project and I feel it was a learning experience that I took to well and enjoyed throughout.

## Conclusion

FindMyTeam is a large full stack software project that took a large amount of project management to complete competently. Without the project management done by everyone involved this project would not have been possible and certainly would not have been as successful as it is now. Project management is an invaluable part of any software development project and was handled well throughout the project, enabling quick, efficient and successful development to occur without any major issues encountered along the way.

# Conclusion

FindMyTeam is a web application designed to facilitate the connection of different users that play the video game League of Legends, this is done through a system of matching similar to that found in dating applications, with a swipe or carousel feature included displaying users to one another and allowing them to like each other. This application is meant to solve the problem of poor team composition within video games especially in ranked situations, where the team composition and skill level is directly correlated to a win condition within the game. With players being able to find each other on FMT this will allow them to create their own teams, and maximise their composition and competitive advantage.

FindMyTeam used many technologies during its development, these include SQLLite (development) and PostgreSQL (deployment) for database management, Python Flask for back end development, ReactJS-Vite for front end development and ShadCN and TailwindCSS for styling and user interface design.

Research was conducted around convolutional neural networks, deep learning and their place within recommender systems. This section informed the basis of understanding for FMT especially around how and why these systems are used in similar applications and their shortcomings. Eventually leading to decisions taken within FMT’s development which shaped how it would function moving into development.

The design phase of the process helped to create a vision for FindMyTeam’s future, how the application would look, what kind of functionality it would need to achieve its goal, how it would fill its gap in the market and what people would need to be taken into account when developing such an application. This stage was imperative to get right and set the groundwork for FMT’s development during implementation.

Implementation was the largest phase of development, here the application was created from the ground up using the technologies decided within the proposal, the information gathered during the research phase and the vision developed during the design phase. All of this was then put together during the implementation phase. Sprints were held each week between the developer and supervisor to debrief the week gone, and plan the week ahead. Through months of work implementation was completed and FMT was deployed using Render for the back end and database, and Vercel for the front end.

Testing was done throughout the development of the project, especially in regards to functionality where ever route, page and function was tested for each possible use case. Error handling and correct response data were tested rigorously to enable users to utilise FMT without being met with errors and issues.

Once the implementation phase was almost over, user testing was done to give an understanding of the parts of FMT that were lacking, either in fidelity or in function. A number of issues were found during this phase and changes were made to FMT based on these issues and user feedback given by the participants.

Putting all of this together FindMyTeam ended up a fully designed and functional full stack application that accomplishes its goal of connecting users through a matching system to play video games together competitively. Through API ingestion from third party apps players have a myriad of statistics and data to pour over for each possible connection, of those they can filter through based on their needs and preferences. And once they decide on a user, matching and connecting is simple, easy and quick.

Without the project management that occurred throughout the project none of this would have been possible, it enabled communication and work ethic to help develop this application in a procedural and efficient manner, concluding with the application being developed to its best state.

During the process of proposing, researching, developing, implementing, testing and managing FMT multiple skills both soft and hard were developed and gained, from new programming languages to a better understanding of professional structure and communication this project was a massive learning experience.

Further development of this project would consist of many different facets. Gathering a base of data and adding a recommender system into the user stack search would be the biggest change, allowing for an increased sense of tailoring for each user that utilises FMT. Smaller developments would include the inclusion of profile pictures, more user customization, the ability to ingest more data from RIOT through a production level API key which isn’t available for personal or educational purposes. These things would help FindMyTeam to become a fully commercial level application which could host many users and facilitate many more connections.

# References

AndreaB. (2021, June 22). *What is Scrum? | The Agile Journey with PM-Partners*. PM Partners. <https://www.pm-partners.com.au/insights/the-agile-journey-a-scrum-overview/>

Schafer, J. B., Konstan, J. A., & Riedl, J. (2001). E-commerce recommendation applications. *Data Mining and Knowledge Discovery*, *5*, 115–153. <https://doi.org/10.1023/a:1009804230409>

Batmaz, Z., Yurekli, A., Bilge, A., & Kaleli, C. (2018). A review on deep learning for recommender systems: challenges and remedies. *Artificial Intelligence Review*, *52*(1), 1–37. <https://doi.org/10.1007/s10462-018-9654-y>

Zhang, Q., Lu, J., & Jin, Y. (2020). Artificial intelligence in recommender systems. *Complex & Intelligent Systems*, *7*(1), 439–457. <https://doi.org/10.1007/s40747-020-00212-w>

Li, Z., Liu, F., Yang, W., Peng, S., & Zhou, J. (2021). A Survey of Convolutional Neural Networks: Analysis, Applications, and Prospects. *IEEE Transactions on Neural Networks and Learning Systems*, *33*(12), 1–21. <https://doi.org/10.1109/tnnls.2021.3084827>

Mahesh, B. (2018). Machine learning algorithms - a review. *International Journal of Science and Research (IJSR) ResearchGate Impact Factor*, *9*(1). <https://doi.org/10.21275/ART20203995>

Deng, L., & Yu, D. (2014). Deep Learning: Methods and Applications. *Foundations and Trends® in Signal Processing*, *7*(3-4), 197–387. <https://doi.org/10.1561/2000000039>

Middleton, M. (2021, February 8). *Deep Learning vs. Machine Learning — What’s the Difference?* Flatiron School. <https://flatironschool.com/blog/deep-learning-vs-machine-learning/>

Salakhutdinov R, Hinton GE (2009) Deep Boltzmann machines. *Proceedings of the 12th international conference on artificial intelligence and statistics, Clearwater Beach, Florida, USA, vol 1,* 448–455

Hinton, G. (2009). Deep belief networks. *Scholarpedia*, *4*(5), 5947. <https://doi.org/10.4249/scholarpedia.5947>

Hinton, G. E., & Salakhutdinov, R. R. (2006). Reducing the Dimensionality of Data with Neural Networks. *Science*, *313*(5786), 504–507. <https://doi.org/10.1126/science.1127647>

Sarwar BM, Karypis G, Konstan JA, Riedl J (2001) Item-based collaborative filtering recommendation algorithms. *Proceedings of the 10th international conference on world wide web*, Hong Kong, China, 285–295.

Van Meteren R, Van Someren M (2000) Using content-based filtering for recommendation. *Proceedings of the workshop on machine learning in the new information age*, Barcelona, Spain, 47–56.

Lops, P., de Gemmis, M., & Semeraro, G. (2010). Content-based Recommender Systems: State of the Art and Trends. *Recommender Systems Handbook*, 73–105. <https://doi.org/10.1007/978-0-387-85820-3_3>

Tran T, Cohen R (2000) Hybrid recommender systems for electronic commerce. *Proceedings of the AAAI workshop on knowledge-based electronic markets*, Austin, TX, USA, vol 4

Zainurrohman, A. (2021, February 27). *Content-Based Recommender System Using NLP*. Medium. <https://arif-zai-nur-rohman.medium.com/content-based-recommender-system-using-nlp-445ebb777c7a>

Loh, W.-Y. (2011). Classification and regression trees. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, *1*(1), 14–23. <https://doi.org/10.1002/widm.8>

*How can you calculate Linear Regression? - Voxco*. (2021, December 7). Voxco. https://www.voxco.com/blog/how-can-you-calculate-linear-regression/

Fitzmaurice, G. M. (2016). Regression. *Diagnostic Histopathology*, *22*(7), 271–278. <https://doi.org/10.1016/j.mpdhp.2016.06.004>

*Regression model: Definition, Types and examples - Voxco*. (2021). <https://www.voxco.com/blog/regression-model-definition-types-and-examples/>

James, G., Witten, D., Hastie, T., Tibshirani, R., & Taylor, J. (2023). Linear Regression. *An Introduction to Statistical Learning*, 69–134. <https://doi.org/10.1007/978-3-031-38747-0_3>

Cunningham, P., & Delany, S. J. (2021). k-Nearest Neighbour Classifiers - A Tutorial. *ACM Computing Surveys*, *54*(6), 1–25. <https://doi.org/10.1145/3459665>

Louppe, G. (2014, July 28). *Understanding Random Forests: From Theory to Practice*. ArXiv.org. <https://arxiv.org/abs/1407.7502>

IBM. (2021, November 2). *Decision tree*. Ibm.com. <https://www.ibm.com/think/topics/decision-trees>

Nader, K. (2020). DATING THROUGH THE FILTERS. *Social Philosophy and Policy*, *37*(2), 237–248. <https://doi.org/10.1017/s0265052521000133>

Schafer, J. B., Konstan, J., & Riedi, J. (1999). Recommender systems in e-commerce. *Proceedings of the 1st ACM Conference on Electronic Commerce - EC ’99*. https://doi.org/10.1145/336992.337035

‌