**A Level Computer Science NEA:***By Harry Espley*

**1 – Analysis**

**1.1 Background**  
**1.1.1 Problem Statement:**Fantasy Premier League (a.k.a. FPL) is the official “fantasy football” game, which is played by over 11.5 million players across the globe as of 2023. The game is owned and operate by of the Premier League and has been running since the 2002/03 Premier League season. I would like to create a streamlined alternative to the current system, with new quality of life features added. Participants select a team of real-life footballers and earn points based on their performance on the pitch. Every participant (a.k.a. “manager”) has a given budget to spend on players and must make strategic, informed decisions each week about a variety of factors, including transfers, team selection and captaincy. As the aim of FPL is to accumulate as many points as possible over the course of the season, millions of enthusiastic participants worldwide constantly seek new and exciting strategies to optimize their teams. While the FPL app does provide a variety of tools to remain informed on the best players – I believe that there is a gap when it comes to specific, tailor-made recommendations based on the participants own team. Players can spend vast amounts of time simply researching what transfers could be made – and in the end it could all be for nothing when the player doesn’t perform as expected. Due to these facts, I aim to create and FPL companion app that not only has most of the same features that the base app has, but also eases the experience further by being able to recommend transfers, optimal lineups and captaincy choices among other things.   
**1.1.2 Problem Context:**  
For the reader to have the most complete understanding of the problem at hand, there are some pertinent issues regarding FPL that must be discussed.   
For the reader to have the most complete possible understanding of the systems and algorithms that will exist within my solution, and why and how they were written; there are some pertinent issues regarding FPL that need to be discussed.   
**1.1.2.1 Team Selection:**At the start of every Premier League campaign, FPL gives every player in the Premier League a value (with a minimum of 4.0 and a record high of 14.0), this value can change throughout the season based on how many people are transferring that player in/out of their teams. Also, at the start of the season, every participant is given a budget of 100.0, from which they must select a team of 15 players, with the following number of players in each position.  
{Goalkeeper: 2, Defender: 5, Midfielder:5, Forward:3}   
Players may also not have more than 3 players from the same team. Similarly, to player values, player positions are decided by FPL at the start of the season, and do not change over the course of the season – even if a player is being played in a position which does not match their position in FPL.   
Transfers in FPL, are when a participant trades one of the players in their squad, for another player in the FPL database, in order for a player to be able to be transferred in:   
***Total buying cost of players transferred in =< Spare Budget + Total selling cost of players transferred out***  
With rounds of Premier League fixtures organized into “gameweeks”, before the start of the first gameweek, unlimited transfers can be made – however each gameweek has a deadline, where after that point no more transfer can be made – after the deadline for Gameweek 1, a participant no longer has unlimited transfers. A participant acquires 1 free transfer per gameweek, with a maximum of 2 free transfers being stored by a participant at any given time. Any transfers made by a participant will be applied upon the deadline of the next gameweek.   
In terms of the rules regarding a user’s lineup, just like football, a participant may only start 11 players at once, with these being the only 11 players who the participant receives points from. They can be any combination of a participants 15 players if they adhere to the following rules.   
**Goalkeeper:** Minimum of 1, Maximum of 2  
**Defender:** Minimum of 3, Maximum of 5   
**Midfielder:** Minimum of 2, Maximum of 5,   
**Forward:** Minimum of 1, Maximum of 3  
Players can also select a **Captain** and **Vice-captain** for their lineup, any points that a player with captaincy scores, will be counted as double (so if your captain scores 8 points in a given gameweek you will receive 16 points in return). However, if the player that you captained does not play a single minute of football in that gameweek, then the captaincy effect will be applied to the vice-captain of your team, this is the only function of the vice-captain.  **1.1.2.2 Points Scoring:**In the regular points scoring system, players score points based off several actions that they take upon the pitch:

|  |  |
| --- | --- |
| ***Action*** | ***Points*** |
| For playing up to 60 minutes | 1 |
| For playing 60 minutes or more (excluding stoppage time) | 2 |
| For each goal scored by a goalkeeper or defender | 6 |
| For each goal scored by a midfielder | 5 |
| For each goal scored by a forward | 4 |
| For each assist for a goal | 3 |
| For each clean sheet by a goalkeeper or defender | 4 |
| For each clean sheet by a midfielder | 1 |
| For every 3 shots saved by a goalkeeper | 1 |
| For each penalty saved by a goalkeeper | 5 |
| For each penalty missed | -2 |
| Bonus points for the best players in a match (see 1.2.2.2) | 1-3 |
| For every 2 goals conceded by a goalkeeper or defender | -1 |
| For each yellow card | -1 |
| For each red card | -3 |
| For each own goal | -2 |

**Clean Sheets:** If a player is substituted after the 60th minute and they had a clean sheet up until that point, they will keep their clean sheet points no matter how many goals their team concedes.  
**Assists:** Assists are awarded to the player that makes the final pass before a goal is scored (whether intentional or unintentional). If the trajectory of the pass is significantly altered by an opposition player before the goal is scored, then there is no assist awarded. If a shot on goal is blocked by an opposition player, saved by the keeper, or hits the woodwork and a goal is scored from the rebound, an assist is awarded to the first shooter. If a player shoots or passes the ball, forcing an opposition player to score an own goal, then an assist is awarded. The player who is fouled to earn a penalty will be awarded an assist, providing he is not the player who takes the penalty.   
The **Bonus Points System (BPS)** utilizes a range of statistics to create a BPS score for every player in each match. The three best performing players in the match receive the following number of bonus points each:   
**{1st: 3 Points, 2nd: 2 Points, 3rd: 1 Point}**   
**1.1.2.3 Chips:**Chips are essentially ‘power ups’ or ‘buffs’ that participants can apply to their team for any given gameweek across the season. However, they must be applied before the deadline for that gameweek has passed. Each chip may only be used once in a season. There are four in total, Triple Captain, Free Hit, Bench Boost and Wildcard.   
**Triple Captain:** Triple Captain means that a participants captain’s points are tripled instead of doubled in the next gameweek.   
**Free Hit:** Allows a participant to make unlimited transfers for a single gameweek, at the next deadline their squad is returned to how it was at the start of the gameweek.   
**Wildcard:** Means that all transfers (including those that have already been made) in the current gameweek are free of charge.   
**Bench Boost:** The points scored by the players on your bench (the 4 players that were not included in your starting 11 from the squad of 15), are added to your total points for the next gameweek.

**1.1.3 Evaluation of current system:**While I deeply respect the expertise of a team of seasoned developers and software engineers – and in no way can I compete with them in most regards – I believe there is room for innovation. My goal is to include distinctive features in my companion app that have been overlooked by the team at FPL. To ensure my app is as feature-complete and comprehensive as possible, the following is an in-depth examination, and in turn evaluation, of the existing system, enabling me to discern its strengths and areas that may require improvement.   
**1.1.3.1 User Interface:**In general, the FPL website embodies numerous user-interface design principles that could be lent upon when creating my own companion app. Its navigation is quite straightforward, owing to a well-structured and logically organized layout allowing users to seamlessly switch between core features. In addition, the design is clean and maintains a consistent aesthetic, allowing users to focus on the core functionalities – however unfortunately there is no option whatsoever for customization of the design. Another aspect of the design that warrants attention, although it is not necessarily a user-interface design flaw, is the extensive display of information and data on the FPL website. Of which the sheer volume and complexity can be overwhelming for new users. Given that my companion app will also feature a wealth of statistics, it’s important that I remember to keep the presentation of said data user-friendly and easy to read/understand.

**A screenshot of a football game

Description automatically generated**

Transitioning to the team-view screen, the design remains clean and consistent. One notable feature is the clear distinction between the players in the starting XI and those on the bench, a feature which I would look to integrate into my user-interface. The inclusion of both a ‘formation’ view and a ‘list’ view, allowing users who have a specific preference to decide for themselves is also a great addition. Additionally, the interactive buttons enabling substitutions (whilst also providing essential data about players e.g., next fixture, injury status...), add a layer of user engagement that I would be keen to implement in my system. Once again, an area where the user-interface falls short is customization, of which there is no option – something I should look to remedy in my system.

**A screenshot of a computer

Description automatically generatedA screenshot of a football game

Description automatically generated**

Finally, delving into the league-view screen, this screen serves as a hub for the leagues that the user is in (whether public or private), allowing them to view the standings of said leagues. The design remains consistent with the rest of the website: clean, streamlined and without too much clutter. Whilst the current display does offer basic, essential data, it could benefit from more information. For instance, being able to see another manager’s MVP (Most Valuable Player) for that week could be an invaluable insight for the user. Moreover, the screen seems to miss out on some ‘just for fun’ or ‘bragging rights’ features – perhaps there could be a ‘Manager of the Month’ badge that appears next to a manager when they have accumulated the most points within a given month. This feature would recognize outstanding performances, whilst also playing into the competitive landscape of FPL.

**A screenshot of a computer

Description automatically generated**

**1.1.3.2 User Experience:**The user-interface’s intuitive navigation significantly enhances the user experience, and the swift data loading ensures a smooth and seamless journey. Moreover, the real-time updates during matches heighten the immersion, making users feel intimately connected to their teams and intensifying the competitive thrill of the experience – as you can see in real time how players actions are affecting yours and other’s teams. However, the presentation of abundant data without sufficient contextual explanation could potentially confound users (especially users who are newer to FPL), slightly detracting from an otherwise streamlined experience.  **1.1.3.3 Features:**The fundamental features that the FPL website includes are:   
- Users are able view how many points you scored, the average points, and the highest points in the current/most recent gameweek.   
- Users can view their team, including information such as injury status or the upcoming fixture for each individual player.   
- Users can view several simple player statistics for every player in the FPL database.   
- Users can see when the next transfer deadline is  
- Users can view the leagues that they are in, the standings of each of these leagues, and their position in said standings.   
**1.2 Research**   
**1.2.1 Survey:**The below section includes the preparation for and the results of a survey that was filled out by members of the FPL community. I prepared questions that I thought would guide me in the design of this system. Most of the questions are oriented around the usage patterns of FPL participants to infer what would be required out of a new system – however I also ask direct questions about what they would require from a new system.   
**1.2.1.1 Survey Preparation:**

|  |  |
| --- | --- |
| **Question:** | **Reason:** |
| Do you play in “mini leagues” with friends or colleagues? | If many people play in mini leagues, it could be important to add features where you can view said leagues. Could also introduce features that allow the user to see how their team compares to others in the league. |
| How often do you currently make transfers in FPL? | Should give a good indication of how often player data will need to be updated on our database. |
| Do you use any “FPL-focused” sources of information to inform your transfer decisions? | To see whether players are already trying to make informed decisions in FPL, and if not, the transfer optimisation could be a welcome feature as it does lots of the heavy lifting for the user. |
| Do you find it challenging to decide which players to transfer in/out? | Helps to indicate whether there is a demand for the transfer optimisation feature. |
| What attributes do you find to be most important when judging a player in FPL? | This question should help in deciding how our system will judge a player’s value. The player rating algorithm will be discussed further at a later point, however the information gathered from this question will be considered when deciding how this algorithm is going to function. |
| Would you rather have transfer recommendations that were personalised to your own team or general recommendations for players to look out for? | Should give a better idea of how the transfer optimisation feature should function, as in whether we must view the users team before making recommendations or not – which of course would take up more API calls. |
| How much budget do you usually leave in your FPL balance? | This question will help us find out how much budget should be left over when transferring players in/out of an FPL lineup. However, if these results are inconclusive then a feature could be introduced to allow the user to decide how much budget they want to be left over instead. |

**1.2.1.2 Survey Results:**   
The following is the results of the survey; it was created using Microsoft Forms and distributed by QR code in person to FPL participants in my school.   
A blue and orange pie chart

Description automatically generated  
Based on our sample of FPL players, it’s evident that most FPL players play in leagues with their friends and acquaintances – rather than just playing in the generic leagues provided by FPL. Due to this trend, it would be beneficial to our system to introduce features that enable users to easily access and view the leagues that they are in. A tool to compare your team to others in that league could also be introduced, however I believe this to be less pertinent.   
A colorful pie chart with text

Description automatically generated  
From the survey results, it’s clear that a significant portion of the FPL community actively engages in player transfers. Whilst a minor segment refrains from making any transfers at all, I feel safe in considering them to be outsiders as their numbers are so limited. In addition, incorporating well implemented transfer features in our app may motivate this minority to participate in player transfers.   
A screenshot of a computer

Description automatically generated  
The survey results indicate that while a notable segment of the FPL community does actively seek out FPL-specific information sources, as expected, the majority do not. Thus, indicating a potential market opportunity for a system of transfer optimization, which our system could capitalize on.   
A screenshot of a graph

Description automatically generated  
Supporting the insights from the previous question, the survey data from this question suggests that a significant number of FPL participants tend to struggle with transfer decisions – reinforcing the demand for transfer optimization tools, affirming the decision to incorporate such a feature into the system.   
A graph with blue squares

Description automatically generated with medium confidence  
In this question, respondents were prompted to rank the listed attributes in order of importance to them when they are evaluating FPL players. This feedback will be instrumental in creating the weighted algorithm that will be employed by our system to rate players. Since this system is being made for the FPL community, their input is invaluable.   
A blue and orange pie chart

Description automatically generated  
The survey results to this question indicate a preference within the FPL community for a personalized transfer recommendation system, specifically tailored to individual teams, rather than a generic one that merely suggests high-performing or undervalued players. This emphasizes the importance of implementing a personalized recommendations feature. Nevertheless, in the interest of ensuring a comprehensive and feature-complete system, I will also plan to implement a more generic recommendation mechanism (in order to cater to the minority that favor it)   
A screenshot of a computer

Description automatically generated  
Whilst the survey results from this question suggest that most FPL users typically maintain a balance of 0.0-1.0 in their FPL account, given the considerable variance in responses, it would be more prudent to allow users to set their preferred balance – in the interest of ensuring a more personalized experience.   
**1.2.1.3 Conclusion:**   
This survey was invaluable, providing a deep insight into the FPL community and helping shine a light on their preferences regarding features for an FPL companion app. The following are some of the key points that I picked up on:   
**Leagues:** Functionality for viewing leagues should most certainly be included and the implementation of a system that allows users to compare their teams to others in their league should be considered.   
**Transfers:** Based on the survey results, there’s a clear demand for a transfer recommendation system, given the challenges many FPL users face in making transfer decisions. Given the frequency with which FPL participants make transfers, it’s imperative that our player data remains as current as possible. While I lack a dedicated server, the solution could be to refresh the database each time a user initiates the program. This approach, involving a straightforward API call followed by database write, ensures accuracy without being resource intensive.   
**Player Rating:** The insights derived from this survey, which highlight the criteria FPL users prioritize when assessing players, will be pertinent in the creation of the weighted algorithm that will be used to rate players.   
**Budget Limitations:** Rather than predetermining the remaining budget for users during transfers or team creation, the system will allow users to set their own budget constraints (within reasonable limits). This approach aims to ensure a more tailored and individual experience for each user.   
**1.2.2 Data Analysis:**  
To offer reliable advice to users regarding FPL, the system should utilize a significant amount of data analysis – ensuring that decisions aren’t made based on “gut instinct” or intuition, but rather on solid evidence and observed patterns. Most of the data will be sourced from the FPL API, (https://fantasy.premierleague.com/api/), which provides a variety of endpoints including areas like player and team data.   
In the system, the data will be represented in an SQL database, with objects being initialized in the code when they are required, using the data from the database to do so. The data from the FPL API is represented in the form of JSON, meaning it will need to be parsed for the relevant data and then committed to my own SQL database as previously stated.   
**1.2.2.1 Player Performance Analysis:**   
In terms of player information and statistics, the FPL API offers a wide variety of information about the players in the FPL database, such as:   
**Player Name:** Including options for “First Name”, “Last Name” and “Web Name” (Web name referring to the name used when representing the player in a lineup).   
**Player ID:** Every player is given a unique ID the start of the season.  
**Team:** The real-life team that a player plays for (is a number 1-20, with each number representing a different team in the Premier League).   
**Position:** Is a number (1-4) representing the position that a player is attributed to in FPL.   
**Form:** The form of a player is a numerical value representing their mean points per match in all the games that their team played in the last 30 days (regardless of whether they played in all those matches or not).   
**Total Points:** The total points are the number of points that the has accumulated over the course of the season.   
**Points per Game:** The points per game is calculated as follows ***Player total Points/Player Games Played***  
**Cost:** The cost of a player is the cost of buying a player on the transfer market.   
**Selling Cost:** The selling cost of a player is the value that a user would receive when trading a player on the transfer market.   
To generate a comprehensive rating for each player, these data points will be integrated into a weighted algorithm – resulting in a single numerical value for each player, their **composite score.** Not only does this streamlined scoring system simplify player assessment for the user – but also empowers our platform to suggest informed transfer and lineup decisions.  
**1.2.2.2 Fixture Difficulty Analysis:**While the FPL API provides a Fixture Difficulty Rating (FDR) each gameweek for every team, I would like to at least contemplate the idea of the creation of my own metric specifically designed around the FPL’s points-scoring mechanics. The primary objective being to evaluate the average FPL points a team tends to concede to individual players, rather than the challenge a whole team faces when playing said team. However, this all depends on whether this is possible using the data provided to us by the FPL API – which will be investigated in the Fixture Difficulty Rating Algorithm section.   
**1.2.3 Algorithmic Design:**   
**1.2.3.1 Player Rating Algorithm:**For the Player rating algorithm, I plan to employ a weighted system. The algorithm will combine various player statistics and generate a “composite score” in the form of a floating-point number. It is also important to note, however, that the algorithm may require multiple iterations. This is because while regular transfer and wildcard suggestions should consider projections for several upcoming matches, free-hit recommendations will need to focus on only the next match. These two separate algorithms ensure that our recommendations are suited to the different strategies employed by FPL participants.   
In practice, a dictionary data structure will be used to store the weights for each relevant player statistic – allowing for an efficient lookup and modification of weights. The following is the essence of this method:   
**composite\_score = sum([getarr(player\_information, key)\*player\_weights[key] for key in player\_weights.keys()])**  
The following is a breakdown of the techniques to be used in this solution:  
1. **Dictionary (player\_weights):** This dictionary indicates the importance/weight of each player statistic.  
2. **getattr():** Usage of the built-in function ‘getattr’ is to dynamically retrieve attributes from ‘player\_information’. This is a powerful tool as it allows us to access the object attributes that relate to the keys in the dictionary ‘player\_weights’.   
3. **List Comprehension:** List comprehension is a python feature that allows us to create lists in a concise and clean manner. In this instance, it is used to create a list of products (each player statistic multiplied by its associated weight), which are then added together by the inbuilt function ‘sum’ to create a total composite score.   
For the algorithm to function seamlessly (without error), it’s important that the key names in ‘player\_weights’ are consistent with the attribute names in ‘player\_information’ – any discrepancy could lead to errors that would need to be handled at runtime.  **1.2.3.2 Transfer Recommendation Algorithm:**When it comes to suggesting player transfers tailored to an individual user’s team, several strategies can be pursued. I have identified three primary approaches, some of which would use the player rating algorithm to assist them, to this challenge which I will detail as follows:   
**Iterative Approach:** Utilizing an iterative approach would entail systematically comparing (using their composite score provided by the player rating algorithm) each player in the FPL database against every individual player in the user’s current team. At first glance, this method may seem to be comprehensive, ensuring that no transfers are overlooked, surely this method must uncover some diamonds in the rough? However, upon further inspection, given the large number of players in the FPL database, this approach rapidly becomes increasingly slow to compute. In addition, not all comparisons would be relevant (comparing a Defender to a Midfielder for instance) meaning that extra logic would be required to be put in place further complicating and decreasing the speed of this solution. The delayed response that this approach would create might be seen as unacceptable to user’s pushing them away from the system – so for that reason I will not utilize an iterative approach when creating this algorithm.   
**Greedy Approach:** An approach grounded in the Greedy Algorithm would not utilize the composite score produced by the player rating algorithm. This stems from the inherent nature of greedy algorithms, as they are designed to pinpoint the best possible option in the immediate present. Within the context of the user’s FPL team, the greedy algorithm would target the most glaring vulnerability – the player/s who accumulated the fewest points in the most recent gameweek. With this information, the algorithm would then parse through the pool of available players, choosing the player/s who scored the most points in the most recent gameweek – if this player still allows the user’s team to adhere to selection regulations. It should be noted that to make sure that the player selected adheres to the FPL team selection regulations, additional logic would have to be appended to the algorithm, ensuring that all recommendations are compliant with the game’s rules.   
Although the Greedy Algorithm would be significantly faster than the iterative approach, there are numerous problems that are caused by it:   
1. **Lack of Future-planning:** The greedy algorithm only focuses on the immediate recent past, therefore it may miss out on players that may have been injured in the past gameweek or have a favourable run of fixtures coming up. Additionally, just because a player performed well in the most recent gameweek, doesn’t mean that they will perform well in the next – or any gameweeks after the next for that matter.   
2. **Potential for impulsive reactions:** By basing transfers simply based on the last gameweek, the greedy algorithm could suggest that the user transfer out a consistent high-performer based on just a single gameweek where they did not perform or didn’t play – this could mean that the user misses out on the high-performers high points tallies in the future.   
In conclusion, although the greedy approach is very fast and it can often make good recommendations that are successful, however it is not particularly suited to the long-term planning that a lot of FPL participants like to employ.   
**Linear Programming Approach:** By utilizing the composite score generated for each player in the FPL database by the player rating algorithm, we can frame this problem as a mathematical model. Where the primary objective is to maximize the total composite score while adhering to a set of constraints. These constraints include:   
**Budget Constraints:** The total cost of incoming players should not surpass the combined cost of any outgoing players plus any spare budget that the user has available.  **Position and Formation Constraints:** In accordance with the FPL’s position restrictions detailed in section 1.1.2.1, any player being transferred out must be replaced by a player of the same position.   
**Team Representation Constraints:** A user’s team can have no more than three players from the same real-life football club. All transfer suggestions must adhere to this rule.   
d  
**1.2.3.3 Optimal Lineup Algorithms:**This algorithm will leverage similar linear programming techniques to those used in the Transfer Recommendation algorithm, however its implementation will be far more straightforward. Since there is no need to consider the user’s existing team – its primary objective is to determine the best possible squad for the next gameweek. Working within the standard FPL team constraints (budget, player positions and team quotas), the algorithm impartially assesses each player in the database to maximize the composite score.   
Once again, I will not delve into the complexities of this algorithm at this point, as will be done in the design section.   
**1.2.4 User-Interface Design:**When it comes to user-interface design the main, pivotal decision revolves around the choice of the GUI framework that I will use to craft my UI. Three leading contenders in the Python GUI framework sphere are PyQt, Kivy and Tkinter. In the ensuing section, I’ll delve into the strengths and weaknesses of each of these platforms and ascertain the most suitable choice for this project.   
**1.2.4.1 PyQt6:**PyQt6 is an implementation of the Qt framework for C++, used for creating graphical user-interfaces and applications.   
**Advantages:**- Supports all major desktop operating systems, allowing development of applications that can be run on multiple platforms seamlessly.   
- Allows for a high level of customization in the user-interface.  
- There exists a lot of support and documentation online for both the Qt framework and the PyQt6 library.   
- Has a native look and feel, leading to a more responsive and efficient user experience.   
**Disadvantages:**- A commercial license is required for non-open-source applications.  
- Harder to integrate into a mobile app.   
- Does not receive updates as frequently as other UI-frameworks.  **1.2.4.2 Kivy:**Kivy is a python UI-framework that allows for the cross-platform development of GUI apps – although mainly used for developing multitouch applications. This fact makes it well-suited to creating interactive applications across a variety of platforms.   
**Advantages:**- Kivy applications can be run on a variety of different platforms with little to no changes, allowing a larger audience to be reached.  
- Multitouch is supported out-the-box allowing for developing modern, sleek applications.  
- Kivy has an active and supportive community, meaning troubleshooting and getting help online should be a simple task.   
- Kivy is GPU-accelerated, meaning it utilizes the power of the GPU, resulting in smoother and more responsive graphics – providing a more seamless experience.   
**Disadvantages:**- Performance can be slightly worse than in native applications (such as those made by using Tkinter or PyQt6).   
- Kivy is also slightly harder to learn than other python GUI frameworks, meaning it would take more time to develop using Kivy.  **1.2.4.3 Tkinter:**Tkinter is the base GUI framework for python, providing developers with a rudimentary toolkit to create simple graphical user-interfaces. The Tkinter library is included with all python installations.   
**Advantages:**- Straightforward and easy to learn, potentially cutting development time.   
- Does not require any installation of new modules or libraries as it is included with every python installation.  
- Supports all major desktop operating systems.  
- Non-performance intensive, allowing for a responsive and lightweight application.   
- Benefits from a wide community to offer support and extensive documentation.  **Disadvantages:**- Limited number of features and options when compared to other libraries, with a more limited set of a widgets available to the developer.   
- The look of Tkinter is slightly outdated, meaning it is not the best choice for developing modern applications.   
- Less options for customization than other GUI applications, so an application made using Tkinter may look more generic than an application made with another UI framework.   
**1.3 Requirements**  
**1.3.1 Resource Requirements:**In order to ensure a smooth user experience, the following is required:   
- A **secure internet connection** is required to ensure consistent access to the FPL API and the system’s database.   
- **Python 3.6 and the included libraries** are also required, however this can be provided in the installation process (e.g., via Github)   
- In terms of **memory**, it is recommended that users have at least 1GB of free RAM, although the actual consumption may be significantly lower.   
- An initial **free disk space** of 1GB is recommended for the installation and operation of the application, as this is the standard amount of disk space required for python and the subsequent libraries I intend to use.   
- **Basic graphics processing** is also a requirement; however, this should not be a problem for most modern computers.   
**1.3.2 User-Oriented Requirements:**The following requirements have been decided upon based on the conclusions I have drawn from the research conducted in 1.2 and my own personal experience and grievances with the base FPL website. In no specific order users of the system should be able to:

|  |  |
| --- | --- |
| Requirements | Reasons |
| Users must be able to log in to their FPL account using their email and password | For personalised lineup and transfer data to be accessed, the request must be sent with the cookies that show that the user has logged in. |
| Users must be able to view their current rank in the ‘Overall’ league in the main window | This is a feature that is available in some alternative solutions, which is more convenient than having to check the overall league’s standings, so this will be implemented. |
| Users must be able to view their current FPL team | This is a feature that is available in the actual FPL app and near all alternative solutions, this is a necessary feature for the application as it is essentially what this application revolves around (lineup and transfer optimisation). |
| Users must be able to view the optimal lineup for the next gameweek | Users must be able to view the optimal lineup for the next week, so that they can inform their own transfer and lineup decisions if they do not want to put complete trust into the transfer and lineup suggestion algorithms. |
| Users must be able to request to be recommended transfers (and be supplied with recommendations in turn) | This is the main, headline feature of the application, and so it is a requirement that it be present in some capacity; in order to provide the best possible experience for the user. |
| Users must be able to view all the leagues that they are in, how many players each league has, the standings of each league, and their rank in said standings | This is a key feature that exists in the actual FPL app and most alternative solutions, making it a necessity for the application. |
| Users must be able to access the player database and must be able to sort and filter the database according to a variety of metrics (including composite score) | If users do not want to put complete trust into the optimal lineup and transfer recommendation algorithms, then they can view the data used in these algorithms to better inform their own transfer decisions. |
| Users must be able to change the colour scheme, font used, and the sound buttons make (and the volume of the button’s sound) | Customisation of the user-interface is a key feature that is missing the FPL app and most alternative solutions, therefore implementing it into my application is a key feature that differentiates it from the alternatives. |
| Users must be able to exit the application without clicking the ‘X’ that is in the top right of the application by default | I want the application to full and whole, requiring it to be able to be closed without the use of native operating system tools. |

**1.4 Planning**   
**1.4.1 Risk Assessment:   
1.4.1.1 Data Security Risks:**Given that the application will likely handle sensitive user information, including email addresses and passwords – which unlock the user’s FPL account – it is vital uphold strong security protocols. This can be achieved by not storing raw, unencrypted data. To fortify the data that is being stored, I plan to use advanced security techniques, such as cryptographic hashing and the addition of salting – to ensure user information is not vulnerable to data breaches. I will go more in depth into the hashing algorithms I plan to utilize in the design section.  **1.4.1.2 API Issues:**Due to my systems reliance on the FPL API (a third-party service), there is an unpredictable nature regarding the API’s uptime and reliability. Should the API face interruptions in service, it is pivotal that the system manages said disruptions with grace, and minimal impact on the user experience. As a contingency, the systems database can be used as a fallback, providing data that is not necessarily as current and up to date as it would be from the API, but remains usable in the absence of data from the API.   
**1.4.2 Time Management:**Effective time management is paramount for success in any long-term project. In this project, I intend to use the Agile development methodology, with a special emphasis on the concept of sprints. Not only aiding me with efficient time allocation, but also ensuring the system produced is of the desired quality and functionality. Agile development is iterative and embraces the idea of incremental improvement, ensuring that at any given point in the development of the software, it remains potentially deployable. Through the integration of sprints – a period within which specific tasks must be completed – I have a structured approach to development. Each sprint offers an opportunity to reflect on the code that was written in the sprint, this is because I plan to undertake bi-weekly sprints. This means that every other week, the work I will be doing is mostly making sure the code is efficiently written and there are no unexpected bugs within the system.   
**1.4.3 Ethical and Legal Considerations:   
1.4.3.1 Ethical:**In terms of ethical considerations, there are numerous possibilities and scenarios that need to be considered:   
**User Consent:** It is crucial to obtain user consent prior to collecting, processing or storing their data. Data collection should also be minimized to essential information only, and stringent security measures should be put in place to prevent unauthorized access.   
**Misinformation:** To prevent misleading users, it is imperative that all FPL-related data displayed is current and accurate. If real time updates are not feasible, it must be explicitly communicated to users that the presented data may not be the latest or entirely accurate.   
**Transparency:** For the sake of clarity and trust, the system should explicitly outline its differences from the official FPL website, including any shortcomings or features it may lack in comparison.  **1.4.3.2 Legal:**In terms of legal considerations there are several to consider:   
**Data Protection Laws:** The system must comply with the Data Protection Act (2018), and as such, clear documentation must be provided to users of the system detailing to them how their data will be used, stored and protected.  
**Intellectual Property:** It must be insured that the content in the system is not infringing on any trademarks, copyrights or intellectual property. Due to this, my system will not use any imagery from Premier League clubs such as logos or kits.   
**Third-Party APIs:** Leveraging the FPL API requires a strict following of its terms of service. Excessive requests to the API could lead to access being revoked, for this reason I will ensure that API calls are kept to a minimum, to ensure consistent and sustainable API access.   
**Age Restrictions:** In alignment with the Premier League terms of service, individuals under the age of 14 are prohibited from registering an account. To ensure compliance with these regulations, I will prompt users to provide their age upon their initial user of the program.

**2 – Design**  
**2.1 Introduction**The contents of this section centers both on the visual design and fundamental mechanics of my system. This will be of use to me as it will supply me with a template from which I can successfully satisfy the requirements of my system – as they were stated in Section 1.3.   
**2.2 Chosen Solutions   
2.2.1 Programming Languages:**My system’s core programs will be made using Python3, leveraging my familiarity with the language to enhance the development process. In addition, Python’s variety of user-interface frameworks make it an ideal choice for creating native user-interface applications. This approach eliminates the necessity to learn new languages, libraries or and frameworks from scratch, streamlining the development cycle, allowing for more efficient progression.   
In addition to Python, I will be utilizing SQL (Structured Query Language) to efficiently access and manage the system’s database. This choice being crucial for the storage, retrieval and analysis of both player and team data – due to SQL’s robust and efficient data handling capabilities. **2.2.2 User-interface framework:**For the user-interface framework of my application, I have elected to use PyQt6, namely for its native yet modern appearance, coupled with its extensive customization options. My decisions stems from a personal preference for PyQt6’s visual style when compared with Tkinter and Kivy, as well as its richer support network, comprehensive documentation, and wide array of tools available online. These attributes will be crucial to a more efficient development cycle, aiming to enhance the development experience and the quality of the finished product.  **2.2.3 Database:**In selecting a database for my project, I’ve opted for Microsoft’s SQL Server, accessed and managed via SQL Server Management Studio (SSMS). This choice being primarily driven by its local deployment capability, providing a more cost-effective solution when compared with cloud-based alternatives. Additionally, the inherent flexibility of SQL allows for the easy migration of data to a cloud-based solution in the future, should a public rollout of the application deem this necessary. These two main advantages of cost-efficiency and scalability make SQL Server an ideal choice for the data storage needs of the application.

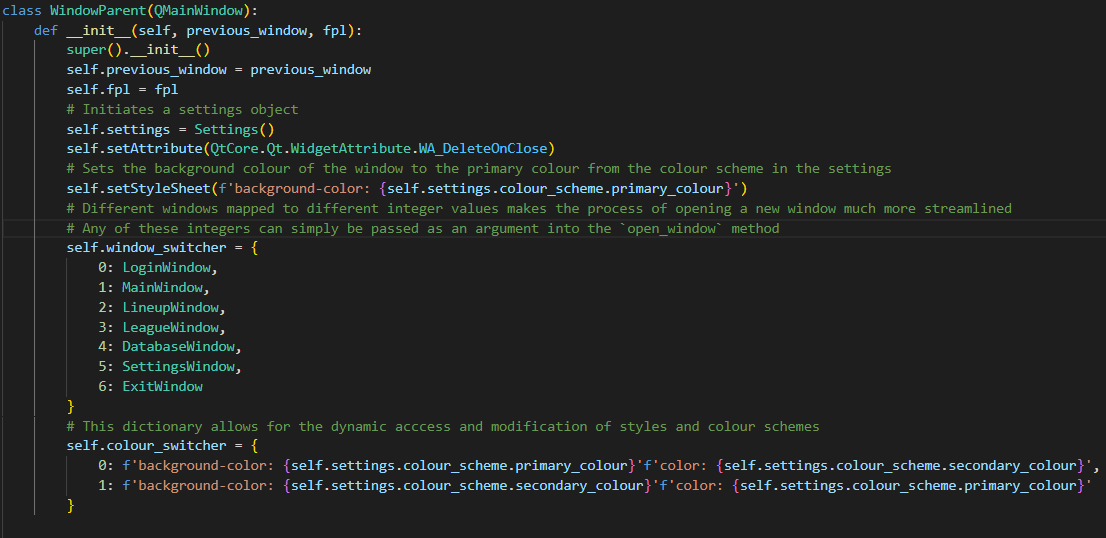
**2.3 System Architecture  
2.3.1 UI Architecture:**The initial interface presented to users upon accessing the application will be the login page. From here, users can authenticate using their FPL account information, with the implemented login features. Upon successful logon users will be redirected to the main page of the application.   
The main page servers as the centralized hub of the application, providing an intuitive user interface for users to transfer seamlessly between the apps core functionalities and features. In addition, an Exit Page is accessible from both the Main Page and Login Page, providing a straightforward option for users to exit the application.   
The application is designed with three core windows, each dedicated to one of the system’s main functionalities. The first, the Lineup Page, enables users to not only view their current lineups, but also request lineup and transfer recommendations to optimize their FPL experience. The second, the Leagues Page, offers users the ability to monitor the leagues they participate in and their standings. Finally, the Player Database page serves as a comprehensive, but also abstracted, player statistics resource allowing users to be better informed when making their own transfer or lineup decisions. This structured approach ensures users possess ease of access to each of the application’s core functionalities – enhancing the user experience.   
The below is a user-interface flow diagram to properly depict the system: A diagram of a company

Description automatically generated **2.3.2 Database Architecture:**My database will consist of two tables, the players table and the teams table:   
**Players Table:** This table is designed to catalog comprehensive data on individual players. It serves as a vital source of data for the algorithms within our system, ensuring efficient and effective operation. The table is structured to facilitate quick storage and retrieval of player information – providing the backbone for player analysis and lineup creation capabilities.   
**Teams Table:** This table is strategically designed to complement the players table, providing information on each real-life football team in the Premier League, of course encompassing all the teams that players play for – and by extension all the teams that players will play against. A pivotal aspect of the teams table is the way it interacts with the players table. The primary key of the teams table ‘Team ID’ is used as a foreign key in the players table, allowing for seamless retrieval of the Fixture-Difficulty-Rating (FDR) for each player, contingent on their respective team. By incorporating this ‘FDR’ into the Player Rating algorithm, the algorithm gains enhanced capability to deliver contextually aware evaluations of player performance, aswell as ‘predictions’ for future performance.   
A screenshot of a computer program

Description automatically generated

**2.4 Data Structures**In my solution I will utilize a variety of data structures, including the following:   
**Array:**   
In my system, arrays will serve as the primary data structure for storing player objects (which are initialized from the player database). By having a structured collection of player objects the Player Rating algorithm, Transfer Recommendation algorithm, and Optimal Lineup algorithm are provided with the dataset that they require to function.   
**Dictionary:**   
In my solution, I will leverage dictionaries to simulate the functionality of ‘SWITCH’ statements, which are not provided natively by Python. By mapping distinct values to respective method calls within a dictionary, I can streamline the execution of different pieces of code based on user input. This method provides me with a clearer and more efficient approach – compared to using nested selection, which could potentially improve the performance of my solution.   
In addition to emulating ‘SWITCH’ statements, dictionaries will also play a crucial role in the algorithms used for calculating composite player ratings. Each attribute determining this rating will be assigned a ‘weight’ in a dictionary, this weight representing the attributes influence on the overall rating. The sum of the multiplications of each attribute’s value and the corresponding weights will provide me with a rating for a given player. Using a dictionary allows for the weights to be dynamically updated, which could be important when tailoring the user experience. Below is a mock-up of what this A computer screen with numbers and symbols

Description automatically generatedsolution could look like:   
I will also employ dictionaries to manage the URLs used when interfacing with the FPL API. Each API endpoint being represented by a unique keyword in the dictionary. This approach allows me to quickly and easily access the URLs without needing to recall or refer to a longer web address – as I will be using multiple endpoints throughout the system. It also enhances readability throughout my code as it is more concise and cleaner, the specific keywords also could provide the reader with a more intuitive understanding of what each part of the code is doing. Finally, having a more centralized storage of API endpoints, the process of updating said endpoints is much faster and simpler – as only the dictionary needs to be updated and not every reference to the endpoint throughout the code.   
**JSON:**Although not strictly a data structure, I feel it is important to mention how JSON will be used in this system. In the system, JSON plays a pivotal role in the storage of user-interface settings – including a range of personalization options such as color themes, font styles and button sound preferences. Shown A screen shot of a computer code

Description automatically generatedbelow is a sample representation of how said JSON might be structured:  **2.5 Algorithms   
2.5.1 Optimization Algorithms:**[EXPLANATION]   
**2.5.1.1 Transfer Optimization Algorithm:   
2.5.1.2 Optimal Lineup Algorithm:   
2.5.2 Rating Algorithms:   
2.5.3 Sorting Algorithms:**In the `Database Window` of the application, users will be afforded the ability to sort the players in the database based on various key metrics. To integrate this feature efficiently, a quick and robust sorting algorithm is of paramount importance. Due to these requirements, I will implement a merge sort, as it is known for its predictable and reliable performance in addition to its ease of implementation. This choice enhances the user-experience by providing quick and accurate data organization.  **2.5.4 Database Filtering:**To enable users to effectively filter the database by various key metrics, several methods can be implemented:   
**Linear Search with multiple criteria:** Implementing a Linear Search with multiple criteria, though not the most efficient, could adequately serve the system’s needs. This method entails incrementally scanning through the player dataset and identifying the players that meet the specified criteria – with these results being compiled into an array. This array is what will then be displayed in the `Database Window`. This straightforward approach should ensure that all player data is presented accurately and comprehensively.   
**SQL Querying:** To implement filtering through SQL queries, the process typically involves a straightforward `SELECT` statement, structured similarly to the following:   
`SELECT \* FROM tbl\_players WHERE criteria1 = value1 AND criteria2 > value2;`   
This method capitalizes on SQL’s powerful data retrieval capabilities, resulting in it being highly efficient for individual filter operations and ensuring that only relevant player data is returned. However, it is pertinent to consider the implications of repeated use of this feature, with the repeating execution of queries which could potentially impact performance. This is an eventuality that will be explored further during the development process.  **Data-management Libraries:** The Python programming language offers a suite of data manipulation libraries, notably `pandas`, which could potentially significantly streamline the integration of database features into the `Database Window`. Utilizing pandas would first involve transferring the data from the database into a `Pandas Dataframe` - a two-dimensional, tabular data structure which is optimized for data manipulation. This transfer can be easily facilitated through the user of SQL database-specific connectors provided by external libraries.   
Once the data is formatted into a Pandas Dataframe, pandas’ powerful built-in functions can be employed to filter it, ensuring that only pertinent player data is presented to the user. Pandas stands out for its exceptional ease of use and in-memory processing, which would obviously not be available using SQL queries. However, i0t is crucial to acknowledge that to employ pandas would mean that extra steps would have to be undertaken to appropriately format the data to ensure seamless compatibility with the library – this being the main drawback that stems from its usage.  **2.6 User Interface Design**The principles of the user-interface design of this application will be to aim to be consistent, free from clutter and customisable, to improve upon the functionality of the design provided by the FPL website. Different classes will be used for each window, with a parent class determining a common set of attributes, behaviours and methods that all windows in the application will share – in the aim to create consistency and ease of development.  
**2.7 Input and Output**The only user inputs into this program will be the username and password of their FPL account on the login screen and their preferred settings on the settings screen. From this, all the outputs of the application can be provided – without the user needing to enter in any more information or prompts, simply pressing buttons. Possible outputs that the program can provide using this data include:   
- User’s current rank  
- User’s Lineup   
- Recommended lineup for the next gameweek  
- Recommended transfers   
- The standings and names of all the leagues that the user is in  
- Statistics on every player in the FPL database.   
**2.8 Error Handling**As error handling is a crucial aspect of any software development, the importance of the program being able to handle unexpected situations and errors with grace cannot be understated. The programming language I am using, Python, provides robust features to handle errors – of which I will primarily be using `try-except` blocks, which will allow me to catch and handle exceptions with ease. In addition to this, pythons typing system allows me to specify and enforce data types, enhancing the readability and maintainability of my code.   
**2.9 Security**Since as of currently, my system only uses user data temporarily (when retrieving the session cookies for a log on), there is not much risk of the stealing of user data. However, in the future, it may be required for a ‘remember me’ feature to be implemented – meaning user passwords must be stored in a database. Whilst such a feature isn’t explicitly planned at this current moment in time, its potential inclusion would greatly simplify user access by eliminating the necessity for repeated logins. However, the implementation of this feature would necessitate securely storing passwords in the system’s database. To achieve this in accordance with the previously mentioned ethical and legal considerations (in 1.4.3), rigorous security measures would have to be utilized. These measures could potentially include a variety of encryption techniques, with the main principle being that user credentials are not stored in plain text. The main encryption techniques that would be used are hashing and salting. The hashing algorithm that I would use in this scenario is Argon2, as it is the only one that I have used in the past (therefore the only one that I am remotely familiar with).   
**2.10 Performance Considerations**In terms of performance, the application should be able to run on most modern low-end systems. Its core is developed in Python, features a non-graphically intensive user-interface, ensuring minimal strain on both the CPU and GPU. However, one potential performance bottleneck is related to memory management, and more specifically the retention of window objects in memory even after they have been closed. This could potentially lead to increased, unnecessary, memory consumption – therefore impacting the applications’ overall performance negatively.   
However, since the PyQt6 library is being used for the UI of the application, the simple and robust solutions for memory management that the library offers can be used to reduce the memory footprint of the application. In fact, this can all be implemented in a single line of code into the parent class of all the windows in the application. As demonstrated by the following:   
  
This line ensures that whenever a window is closed in the application, it is deleted from memory rather than stored until the entire application is closed. This strategy is critical for maintaining the application’s responsiveness and ensuring it can be run-on low-end systems with lower memory capacity.   
**2.11 Testing Strategy**The features of my application will be tested as they are implemented, keeping in mind the requirements that were set out in the analysis section. To reduce the possibility of the user experience being affected negatively, I will attempt to root out as many errors as possible – by changing the code so that such an error is either handled properly or does not occur.   
As I will be the writer of the program, I will be able to white-box test my application, however I will also invite my family, friends and peers to test the program, to find any errors that a new user may experience when trying the system.   
When properly testing my program, I will perform a mix of Unit and Integration testing, and only when each unit works as expected, and all the units seem to work harmoniously will I test the program.   
**2.12 User Documentation**In the preliminary release of my system, there will be a lack of user as I don’t deem it strictly necessary for such a simplistic system. I believe that through intuitive design of my user-interface and the system’s features, any sort of user documentation will be rendered unnecessary.  **2.13 Code Structure and Organization**The application will be structured into distinct, modular components, each of which encapsulating a unique set of functionalities. This design allows common code segments that are shared across the codebase to be updated centrally, reducing the effect of unintended consequences on other parts of the codebase. The architecture of the application will be organized into four separate files as follows:   
**utils.py:**The `utils.py` file is designed to be the central storage of for various utility elements that are essential across the program. This includes constants, such as API endpoints and commonly used functions, like fetch functions and sorting algorithms. As mentioned previously, centralizing shared resources simplifies codebase maintenance, as updates or bug fixes are immediately propagated throughout the rest of the system. The existence of this file is also in compliance with the DRY (Don’t repeat yourself) coding principle as code that is used multiple times throughout the application is written once, and utilized wherever else it was needed in the application. Finally, having a dedicated utilities file allows for easy navigation throughout the codebase, especially since modern text editors allow developers to trace used functions back to their original definition.  **base.py:**The `base.py` file is designed to function as the foundational module of the system, storing key data structures, entity definitions and functionalities critical to the application’s process. The file will encapsulate core elements like data classes for entities such as ‘User’, ‘Player’ and ‘League’, ensuring that the representation of these components is standardized throughout the application – ensuring efficient data management and manipulation. Additionally, `base.py` will include sophisticated algorithms that can be used to optimize player lineups and transfers, based on several key metrics.   
By centralizing these components, and more that have not been mentioned, the development and maintenance of the application becomes much more streamlined and straightforward. Once again, like `utils.py`, this approach is in accordance with the DRY (Don’t repeat yourself) principle, ensuring that the foundational blueprints for entities in the program are stored in a single location – allowing for the dynamic creation of such entities. **fpl.py:**The `fpl.py` file serves as the main module for the operation of most of the features of the application, it is structured to harness the foundational structures and key functionalities defined in `base.py` and `utils.py`. Within the file, the ‘FPL’ class is stored, functioning as the nerve center for all FPL-related functions of the application.   
This class is engineered to facilitate seamless interaction with the FPL API and the application’s own database, leveraging methods and constants from `utils.py` for efficient communication and data exchange with both external and internal services. The key functionalities stored in this class include the retrieval and processing of player information, user lineups and league standings, among others. By centralizing FPL-related operations in this class, the module enhances the maintainability of the codebase – facilitating ease of updates and enhanced readability.  **ui.py:**Finally, the `ui.py` file is crafted to serve as the graphical user-interface (GUI) layer of the application. It utilizes the PyQt6 framework, which is responsible for creating an interactive front-end that facilitates easy navigation to all the applications features.   
The main feature of `ui.py` will be the `WindowParent` class, which will integrate a collection of shared properties and methods that each individual window in the application will require – offering a structured and organized layout throughout the program. Every other window in the application will be inherit from this parent class and will be dynamically generated in the application based on user inputs.   
In essence, the `ui.py` file is the component of the application that brings the visual and interactive elements of the system to life for the user, resulting in an efficient and user-friendly interface.   
**2.14 External Libraries and APIs**My usage of external libraries and APIs will be limited, with the following being used:  **PuLP:** PuLP is a linear programming (LP) modeler for Python, that facilitates the usage of linear optimization within the application. It will be used in both the `TransferOptimiser` and `LineupOptimiser`, to aid the process of suggesting transfers and optimal lineups to the user.  **Mysql.connector:** Mysql.connector is a library used to connect python applications to a MySQL database. It will be utilized in my application to connect to the database that stores player and team information (and potentially user credentials also).  **PyQt6:** PyQt6 is an implementation of the Qt framework (C++), used for creating graphical user-interfaces and applications. It will allow me to create an interactive and sophisticated user-interface for the system.  **FPL API:** The Fantasy Premier League (FPL) API is an unofficial API providing access to the data that the official FPL website and app uses to update the data that is presents. It will be used in my application to provide me with the data to populate my database with player and team data.   
**3 – Technical Solution**The code can be found in its entirety in the appendix of this document (end of the document).  **ui.py:   
WindowParent:**The `WindowParent` class is designed to be a base class for various windows within the application, it inherits from `QMainWindow`, which provides the base functionality of the application window.   
  
The parameters used in the construction of a `WindowParent` object are `previous\_window` - an integer that represents the last window before the current window, the integer being the window\_id of said window – and `fpl` - representing an instance of the `FPL` class. The settings attribute `self.settings` is very important to the customization of the application, as every window having the same settings allows for the user to apply certain settings on one screen, and then have them consistent throughout the application’s many windows.   
The line `self.setAttribute(QtCore.Qt.QWidgetAttribute,WA\_DeleteOnClose)` is a very important aspect of the efficiency of the application, as it ensures that when a window is closed in our application – it is also deleted from memory.   
The dictionary `self.window\_switcher` is what is used to associate each window with a given integer number, so that tasks like refreshing a window or opening a new one can be done seamlessly. Finally, `self.colour\_switcher` is another dictionary, that allows me to associate a certain colour configuration (primary colour, secondary colour and foreground, background) with a certain integer, so that each widget in the application can simply be given an integer value to dictate their colour, rather than me write the same code out many times over.   
A computer screen shot of text

Description automatically generated  
The `refresh` method is a method that is used to refresh the current page, for instance so that any changes to the settings can be made visible. It works by closing the current window, and then creating a new instance utilizing `self.window\_switcher` to associate the `window\_id` of the given window with an actual window class in the program (the same window class that the refresh was initiated from). The refresh method works as when the user applies their preferences in the `SettingsWindow`, the `settings.json` file is updated to reflect said preferences, meaning when the `Settings` object of the new window of the new, refreshed, window is initialized – the new settings are integrated into the properties of the `Settings` object.  
The `open\_window` method of the `WindowParent` class functions similarly to the `refresh` method, however it takes an input of the desired `window\_id`, and hence window, that is to be opened. Additionally, if the `window\_id` entered is equal to 6, (or ExitWindow), then the window that calls for a new window to be opened does not close. However in every other case, the original window closes.  
The operation of the `back\_window` method is actually very simple, it simply checks that there has been a previous window (i.e. the application is not on the LoginWindow), and if there has been a previous window (as indicated by the fact the `self.previous\_window` has a value that is not `None`) then it calls the `open\_window` method with the integer value `previous\_window` as the parameter.   
A screenshot of a computer program

Description automatically generated  
  
  
  
  
  
  
  
  
  
  
  
  
As previously mentioned, each widget in the application has an attribute `colour` which can either be a 1 or a 0, however in order for these colours to be visualized, they need to be ‘applied’ to the widgets themselves. This is done through two methods in the WindowParent class (so that every individual window can apply their own colours), `apply\_colour` and `apply\_colours`. `apply\_colour` checks if the widget chosen (the widget passed as a param has any ‘children’ widgets, and if so, iterates through said children, calling the method recursively on the children widgets. After it has checked if the widget has children, it uses the built-in method `setStyleSheet` for the `QWidget`, along with the stylesheet from `colour\_switcher` that corresponds to the integer value of their attribute `colour`, or in some cases, their parents colour attribute. The `apply\_colours` method, which is called in the initialization of every window, iterates through all the widgets in each window, and if a widget has the attribute `colour` it calls the `apply\_colour` method with the given widget as the argument.  **CustomButton:   
A computer screen shot of a program

Description automatically generated**  
  
  
  
  
The `CustomButton` class is a class used to extend the functionality of the `QPushButton` class from PyQt6, introducing sound effects when a button is clicked. The parameters for the construction of such an object is simply the parent object of the widget, so that the settings, and therefore which sound effect is to be played and at what volume can be accessed.   
In the initialization, if there is a parent widget to the button, then `self.sound\_effect` is set to the .wav file that is in the settings object of the parent object. This is used in the `play\_sound` method, which takes the input `volume` and then plays the sound at that given volume, the volume is dictated by the `button\_sound` property of the parent object. The `handle\_clicked` method is used to link the clicking of the button to the playing of the sound, due to the line `self.clicked.connect(self.handle\_clicked)` in the initialization of the object. The `handle\_clicked` method then checks if there is a parent object, and then calls `play\_sound` for the sound to be played. Creating a custom class to use for buttons also allows the easy implementation of new features for the button, as another method can simply be added to the class and connected in the initialisation.  **Settings:**A screen shot of a computer program

Description automatically generated  
  
  
  
  
  
  
The `Settings` class is responsible for managing the user settings within the application. It reads and writes settings from/to the JSON file `settings.json`, providing methods to change and receive specific application settings.   
In the initialization, the `settings\_path` attribute of the class is set to the path of the JSON file `settings.json`, which is located in the `json\_data` folder of the working directory of the application. After the settings path is set, using `utils.read\_json` (from `utils.py`) and if this fails, it then sets all the values of the settings to `None`; it then writes these default settings to `settings.json` using `utils.write\_json`. However, if the reading of `settings.json` does not fail, then it goes through `settings.json` turning the settings found in `settings.json` into properties of the `Settings` object, with `primary\_colour` and `secondary\_colour` forming a `ColourScheme` object. The property `settings\_switcher` is a dictionary that maps identifiers to the current values of certain settings, allowing for settings to be dynamically accessed and updated.  
The `change\_settings` method allows for the user settings to be updated not only whilst the program is running, but also for the next time that the program is opened open, as it saves the new settings to  
 `settings.json`.   
  
 **ColourScheme:  
A computer screen with text

Description automatically generated**The `ColourScheme` class is a data class designed to store information related to the colour schemes used in the application. It utilizes the `@dataclass` decorator which is used to simplify the creation of classes used for storing data. The decorator does so by automating certain methods such as `\_\_init\_\_` - reducing boilerplate code, therefore making the code easier to understand and modify for future changes and debugging.  
The properties `primary\_colour` and `secondary\_colour` are used to represent the primary and secondary colours of the colour scheme – the colour scheme is linked to the widgets through the settings object. `error\_colour` refers to the colour of the text when an error appears, which would be red, hence error colour has the value `’rgb(255,0,0);`. Certain properties being set to `None` allows for the dynamic styling based on user preferences and runtime conditions.   
**Every Window:**Due to the code being fully commented (and the entirety of the code being provided), I do not feel the need to describe in depth the configuration of the user-interface and widgets of every single window individually. This will generally encompass the `setup\_ui` method of every single window, however if there are key details that need to be explained, such as the function of a certain button or the like, then they will be mentioned.   
**A computer screen shot of text

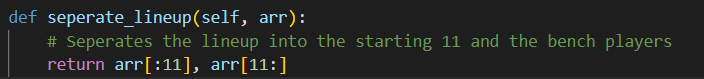
Description automatically generated**  
For each widget in my application, this is the standard configuration. When defining a widget, the item placed in the brackets is the parent of said widget; in this case it is `self` (in order words the window that this widget will be present in). The `set Geometry` method that is used on the widget is used to place and control the size of the object. The first two numbers in the brackets determine its position and the last two control its size. They control it as follows: *(x position, y position, x size, y size)*   
Giving the widget the attribute `colour` allows the `apply\_colours` and `apply\_colour` methods to operate on it and apply the corresponding stylesheet to the `colour` attribute that is in the `colour\_switcher` dictionary in `WindowParent`. The `apply\_colours` method will be called in the initialisation of every window, as it cannot be done in `WindowParent` as the initialisation function of `WindowParent` is called before the UI has been configurated in every window (so `apply\_colours` would have no effect). The method `setText` controls the text that will be on the object if it is able to have it, with `setFont` controlling the type of font used and the size. In all cases in the program, the font used will be `self.settings.font` , which is user controlled, however the size of the font changes based on how large the widget is. Every window class will also use `setFixedSize` in its `setup\_ui` method, this is because it means that the size of the window cannot be modified by the user; if the size of the window could be modified then the widgets in the window would not scale to size – leading to a poorer visual experience for the user. Finally, although the functionality for a button playing a sound when the user clicks it has been implemented, there are no sounds currently in the program files. This is because I was not able to find any ‘button click sounds’ on the internet that did not require payment to use – so if the user wishes they can add their own `.wav` file to the program files, however there is no default sound in this iteration (although this could be added in future development if a suitable sound is found).   
Due to having explained how widget configuration generally works here, I will not go into too much detail about widget configuration in the rest of my explanation of my technical solution (or in the comments of my code).  **LoginWindow:**The `LoginWindow` class represents the window of the GUI application that is dedicated to user login. This window inherits from `WindowParent` whilst also adding its own methods whilst also adding its own methods such as `setup\_ui` and `check\_login`.   
To start, the constructor of the `WindowParent` class is called in the initialization, so that `LoginWindow` can benefit from all of the properties and methods of its parent. Then, a session is created using `requests.Session()`, an object for handling HTTP requests and interacting with the FPL API. If the login is successful, then the session is integrated into new windows using the `FPL` class rather than how it is done here (in the initialization).   
`setup\_ui` is a basic method that creates various UI elements such as labels, buttons and input fields. It then applies the colour scheme to each widget in the window based on their `colour` attribute. The `login\_button` is also connected to the `check\_login` method in this method.  
A screen shot of a computer code

Description automatically generated  
The `check\_login` method is used to use the credentials entered by the user to login to the FPL API, and therefore retrieve the relevant cookies that will allow authorized API requests to be made. First, `utils.get\_account\_cookies’ is used to attempt to login to `fantasy.premierleague.com` - if this does not fail and cookies are present in the return of `utils.get\_account\_cookies` then a user object is created using `utils.create\_user\_object` and an instance of the `FPL` class is initialized – this instance of the FPL class will be passed down to every subsequent window opened in the application. Finally, the `open\_window` method of the parent class is called, to open `MainWindow`. However, if `utils.get\_account\_cookies` were to fail, and the function returned `False` instead of the relevant cookies, then the text of `error\_label` would be changed to ‘Login failed please try again’.  **MainWindow:**The `MainWindow` class represents the window of the GUI application that is dedicated to allowing the user to navigate to the various features of the application. This window inherits from `WindowParent`, whilst also adding its own methods. Whilst `MainWindow` does not have many features of its own, it allows the user to open all the other windows in the application, meaning that it gives the user access to all of the features of the application.  
A screen shot of a computer program

Description automatically generated  
However, `MainWindow does have two features of its own. One being the fact that it shows the user’s rank at the top of the screen. This is done by adding a `QLabel` to the window and setting the text to the function `FPL.get\_current\_user\_rank`, which can be used as all the windows have an instance of the class `FPL`. The second feature of `MainWindow`, is the fact that you can update player data at the press of a button. The `update\_players\_button` is a `CustomButton` object, that when clicked calls `utils.update\_player\_table` and `utils.update\_composite\_scores`, whilst providing the database connection and valid session needed to run these methods. This allows the user to make sure that the data that the system uses to generate lineups and transfers is up to date – which should mean that the suggestions are helpful. `MainWindow` essentially acts as the ‘hub’ of the application, with multiple buttons that take the user to every other window/functionality (except `LoginWindow` as it is not accessible after the user has logged in) in the application. The code for the buttons is shown below, with each integer value passed into the `open\_window` function corresponding to the window that is to be opened in `window\_switcher`:   
**A screen shot of a computer program

Description automatically generated  
LineupWindow:**The `LineupWindow` class represents the window of the GUI application that is dedicated to to displaying the user’s FPL lineup, including the starting 11 and bench players. In addition to this, it is also the place where users can receive tips on chip usage (1.1.2.3) and transfers. `LineupWindow` essentially holds the main functionality for the entire system.   
**A screen shot of a computer program

Description automatically generated**The initialisation of `LineupWindow` is quite complex and takes more time than other windows, as many details, such as the user’s team, user’s balance and the optimal team for a free hit must be calculated before the window is fully opened. It is worth mentioning that the reason the players sell cost is divided by 10 as the value in the FPL API is the players price multiplied by 10; this is the same for any instance of a player’s value being divided by 10 in the codebase.   
The first function in `LineupWindow` is the function used to generate the optimal lineup for the next gameweek, `generate\_free\_hit`.   
**A computer code on a black background

Description automatically generated**The `generate\_free\_hit` function lends on the `ChipOptimiser` class and it’s `@classmethod` `generate\_team` in order to remove the complexity of the code and calculations from `ui.py`. It does so to maintain the readability of this file as UI elements will need to be updated more frequently in future development for the implementation of new features. For the same purpose, the budget to be used on the optimal lineup is calculated using list comprehension and `sum()` functions.   
The next function in `LineupWindow` is quite simple as it splits the full lineup provided by the FPL API (or `ChipOptimiser.generate\_team` into the starting 11 and the bench. It does so use array indexing. With 11 players being in the starting 11 and the remaining (4) players being on the bench.   
****The following method is `setup\_ui`, and although this features in every class, and most of what it contains is simple configuration of widgets, there are some unique features to `LineupWindow` included.   
**A computer screen shot of a program code

Description automatically generated**  
Although the formation view has not yet been created in this iteration of the application, the ability to toggle which view is on the screen at once is included (the `toggle` function will be shown later). So even though when the user switches to the formation view, they are greeted with an empty window, the functionality to switch between views has been implemented already to ease future development.   
Another feature is the fact that the user can seamlessly switch between their own team and recommended teams. This is made possible by the fact that both teams were pre-loaded when the user opened the window (as explained in the section explaining the `\_\_init\_\_` method. This is also made possible using lambda functions, which allow the application to use the same method to change the team shown to either team.   
**A screen shot of a computer program

Description automatically generated**The ability to switch between formation view and list view is enabled by the code at the end of `setup\_ui` and the code inside the `toggle` method, as shown below:   
**A screen shot of a computer program

Description automatically generated**Using a stacked layout on `main\_widget`, the application is able to switch between two different layouts dynamically, with the button corresponding to the layout that is currently active turning green when it is pressed, and being coloured according to the colour scheme when its corresponding view is not active. When toggle is pressed, it simply switches to the layout that it is not currently on, which is made possible by knowing which button was pressed to call `toggle`, which is provided by `sender()`.   
**A computer screen shot of a program

Description automatically generated**  
Since we are using a stacked layout that allows multiple widgets to be shown on `main\_widget`, a layout must be provided to the widget shown. This is how the layout for `list\_widget` is created. This was shown before when the layout of `list\_widget` was set to the return of this function. This layout contains two tables that show both will show both the starting 11 and the bench of the team once it has been populated (any any relevant information).   
**A screen shot of a computer program

Description automatically generated**  
Because `starting\_eleven\_table` and `bench\_table` have been made properties of `LineupWindow` instead of `list\_widget`, they can be modified directly (and as described above). Functions such as `convert\_team` and `convert\_position` are borrowd from `utils.py` to present player data in a readable format to the user (and not as integer values that make no sense to the user).  **LeagueWindow:**The `LeagueWindow` class represents the window of the GUI application that is dedicated to displaying the leagues that the user participates in, whether they be mini-leagues (usually with a small group of friends/colleagues) or competition-wide leagues (with millions of players across the country/globe). Although, functionality to view leagues with millions of players has not yet been implemented (and most likely will not be), as to load that much data without a dedicated server, of which I do not have access to, could significantly harm the performance of the application needlessly.   
The initialisation of `LeagueWindow` is much simpler than `LineupWindow`, as not as much data needs to be collected when the window is first loaded. Instead, only the leagues that the user is currently in have to be loaded. This is done by utilizing `fpl.get\_current\_user\_leagues`, remembering that every window had an instance of the `FPL` class called `fpl`. The actual data for specific leagues is only loaded once a user has selected to view that league, as not as much data needs to be loaded, or calculations made, as numerous players from the FPL database (like in `LineupWindow`).   
**A computer screen shot of text

Description automatically generated**Although `setup\_ui` has not been shown in its entirety when explaining other windows, since `LeagueWindow` has a much simpler user-interface, and a lot of its functionality comes from this method, `LeagueWindow`’s `setup\_ui` method will be shown in its entirety here.   
**A screen shot of a computer program

Description automatically generated**The first widget created `league\_chooser` is a `QComboBox` that allows the user to view all the leagues all the leagues they are in in a compacted format. The window is created so that when the user presses `view\_league\_button`, the league selected in `league\_chooser` will be the league displayed in `league\_table`. Firstly, `league\_chooser` is created and configured, after this the leagues that the user is currently in are iterated through, with the name of each one being added to `league\_chooser`. After this, the `view\_league` button is created, with its function being to populate `league\_table` with the standings of the chosen league when it is clicked – to do this it calls the `view\_league` method which will be explained later. Finally, `league\_table` is created, with the number of columns set as the number of metrics that have been chosen to be displayed for each player in the league’s standings (Rank in the league, name of player, player’s team name, how many points they got in the last gameweek, how many points they have gained throughout the season).   
**A computer screen shot of a program code

Description automatically generated**The final method of `LeagueWindow` is `view\_league`, which is the method that is called when the `view\_league\_button` is pressed. Firstly, the contents of `league\_table` are cleared - in case there was a league already being shown to the user. Then, the league to be displayed is set as the current league selected in `league\_chooser`, with the league id being found by iterating through the leagues that the user is in until the one with the same name as the league chosen in `league\_chooser` is found. Using this id and the `fpl` object of the window, `fpl.get\_league\_standings` is called, which returns a list of JSON strings (for each player in the chosen league). The number of rows in `league\_table` is then set to the length of the array (the number of players in the league and hence the number of rows), with the league then being iterated through. Upon each iteration, the cells in each row are set to the corresponding data found in the JSON string of each player, with the `rank in standings – 1 ` being the row that each set of player information is placed at, as python indexing starts at 0.  **DatabaseWindow:**The `DatabaseWindow` class represents the window of the GUI application that is dediated to displaying in-depth data on every player in the FPL database, including their composite score generated by the application. It provides the user with the option to sort said players by a variety of metrics, in the hopes that it leads to them making more informed transfer decisions in FPL.   
**A screen shot of a computer program

Description automatically generated**In the initialisation of `DatabaseWindow`, one key attribute is `sort\_switcher`, this dictionary will allow the sorting function to sort by certain key metrics, as shown in its contents. The merge sort defined in `utils.py` which is used to sort players for a `DatabaseWindow` instance takes a lambda function as one of its inputs, so when calling it to sort players objects by a certain metric a reference to sort switcher with a given integer value can be passed as its parameter. For example:   
- If we were to pass `key=sort\_switcher[0]` into the merge sort with a provided array of player objects, then the merge sort would sort the players in order of their player id – as the integer value 0 corresponds to `lambda x: x.player\_id` in `sort\_switcher`.   
Another key point is the fact that some of the lambda functions in `sort\_switcher` have a ‘-‘ before the specified attribute, which will make the make the returned array sorted in reverse order (largest value first). This works because a given a negative number of greater magnitude than a negative number of lesser magnitude, the number of greater magnitude will be less than the number with lesser magnitude (-10 and -1 for example). It is also necessary as certain key attributes such as composite score should be sorted with the highest value first, as a higher value in composite score is better.   
After `sort\_switcher` is created, an array of player objects of all the players in the FPL database is generated using the `fpl` attribute of the class and calling `fpl.get\_all\_players`. This will be used as the data to be inputted into the table showing the user all the players.   
**A black screen with white text

Description automatically generated**The first method of `DatabaseWindow` is `sort\_by` which takes the set of players generated in the initialisation and an integer value (corresponding to a key in the `sort\_switcher` dictionary as explained above) indicating what metric the players are to be sorted by. The array is then modified using `utils.merge\_sort` (the functionality of which will be explained later on) with the set of players and metric to sort by as the arguments. Once the array has been sorted, it is returned so that the players can be inputted into the player table.  
**A screen shot of a computer program

Description automatically generated**The second method is similar to `view\_league` in `LeagueWindow`, as it takes the array of players and inputs them into `player\_table`, which is a property of `DatabaseWindow` defined in the `setup\_ui` method of the class. The lone parameter of the method is `value` which is the integer value to be inputted into the `sort\_by` method indicating by what attribute to sort the player objects. Firstly, the contents of `player\_table` are cleared, to remove the prior way in which the players were sorted, as this could conflict with the new order. After the table has been cleared, the raw list of players (unsorted) generated in the initialisation is passed into `sort\_by` along with the value inputted into the method, which returns the same list of players but in a new variable and sorted by the metric specified by `value`. This new list of sorted players is then iterated through as an enumerate object so that `index` can be used to specify the row in which to place the player (as both the iteration and python indexing start at 0) and the value of `x` to represent the player object at index `index` in the array – providing the data needed to populate the table. Functions such as `utils.convert\_position` and `utils.convert\_team` are used to make the positions and teams understandable to the user (as it converts them from the integer form that is present in the FPL API into the string form that would be used in real life e.g. Arsenal or Goalkeeper). The composite score of the player is rounded as often the composite score generated can be a floating-point number with many digits beyond the decimal point, meaning at a point it is redundant to show to the user – only making it harder to read.  **SettingsWindow:**The `SettingsWindow` class represents the window of the GUI application that is dedicated to allowing the user to change the settings of the application such as: colour scheme, font and button sound/volume (if the user has added .wav files to the program files). The initialisation of `SettingsWindow` includes the creation of `settings\_dict`, a dictionary used to store changes in settings. A dictionary has been chosen to represent this as it is the closest data structure in python to JSON, which is pertinent as the settings of the application are stored in a JSON file, `settings.json`. Below is an example of what `settings.json` could look like and the initialisation method of `SettingsWindow`:   
A screen shot of a computer program

Description automatically generated  
A screen shot of a computer code

Description automatically generated  
Because of the structure of `SettingsWindow`, it has to have a different method than the other windows to refresh itself (which will be used after settings are applied to the window). This is since the window contains the widget `main\_widget`, which has a stacked layout applied to it coupled with the fact that the settings have to be changed when the window refreshes; a feature that is present in no other window. The method `settings\_window\_refresh` first checks if `main\_widget` exists and if it does, it deletes it – the program will crash otherwise. The window is then refreshed using the same method as `refresh` but instead using the functions `close` and `open\_window`.  
**A screen shot of a computer

Description automatically generated**  
The next method in `SettingsWindow` is `volume\_value\_changed`, which changes the value of the volume entry when the volume slider is moved, so the representation of the change in value of the value is consistent in both.  
**A black background with white text

Description automatically generated**  
The next 3 methods are all used to populate the `colour`, `font` and `sound` widgets that are added to the stacked layout of main widget in order to show the widgets that let users change the respective settings of each widget. They do not need to be explained in any great detail as they are for the most part simply creating and configuring UI widgets, any explanation required for the functionality of the buttons is included in the comments.   
**A screen shot of a computer program

Description automatically generated**  
**A screen shot of a computer program

Description automatically generated**  
**A screen shot of a computer program

Description automatically generated**  
The next 3 methods in `SettingsWindow` are also related to the colour, font and sound widgets. However instead these methods set the current widget of the stacked layout (which is the layout of the main widget of the window) to each of their respective widgets.  
**A screen shot of a computer program

Description automatically generated**  
Because colour and sound are controlled by dialog boxes, they have to be opened and the data retrieved from them handled and formatted so that the program can interpret them.  
**A screen shot of a computer program

Description automatically generated**  
The above is the methods that open the dialog boxes for both colour and font, their function explained with comments. Both methods were called by `colour\_widget` and `font\_widget` respectively, as shown above, meaning they can only be called if their respective widget is the one being shown – as they are not referenced anywhere else in the code.   
The next methods in `SettingsWindow` are for the sole purpose of updating the `settings.json` file and the `Settings` object of the window.  
**A computer screen shot of white text

Description automatically generated**  
`add\_to\_settings` is used elsewhere in the program to add the users desired settings to `settings\_dict`. The function takes two inputs, a key and a value, so that the key inputted will now be present in `settings\_dict` and will have the value `value`.   
`apply\_settings` is used when the `apply\_settings\_button` is pressed, it calls the `change\_settings` function of the `Settings` object, which changes the settings in `settings.json` (as discussed previously), with `settings\_dict` as its argument. Once the settings have been changed, it then refreshes the window to show the user the new settings in effect.   
The `setup\_ui` method of `SettingsWindow` is mostly the same as other windows in terms of the creating and configurating of widgets, however some functions have to be shown – although there is no need to explain them as the functions they call have already been explained and they are commented.  
**A screen shot of a computer program

Description automatically generated**  
Finally in the `setup\_ui` method of `SettingsWindow`, `main\_widget` and its `stacked\_layout` are created and configured.  
**A screenshot of a computer program

Description automatically generated  
TransferWindow:**The `TransferWindow` class represents the window of the GUI application that is dedicated to showing the user recommended transfers for their FPL team. It is opened in `LineupWindow`. Its initialisation has 3 extra parameters ontop of `fpl` and `previous\_window`, these being `user\_team`, `players` and `budget`. `user\_team` is the team that the user currently has in FPL, so that transfer recommendations can be tailored to their team, `players` is the available pool of players, so that the transfer recommendation algorithm can be provided with a pool of players to compare with the players in the user’s team and `budget` is the balance that the user has in FPL, so that the transfer recommendation algorithm knows how much extra it can spend on incoming transfers. These 3 extra parameters are then used to call `base.TransferOptimiser.generate\_transfers`, after an instance of the class has been created. These transfer recommendations are stored in `self.transfers` so that they can be used to populate the transfer recommendations table.   
A screen shot of a computer code

Description automatically generated  
In the `setup\_ui` method of `TransferWindow`, after the initial configuration of objects, including the configuration of `main\_widget` (which is what will be used to show the table of recommended transfers), the layout of `main\_widget` is set to `self.setup\_list\_view()`, which calls said function which returns a layout that includes the table of transfers, visualising the transfers for the user.   
**A computer screen shot of a program code

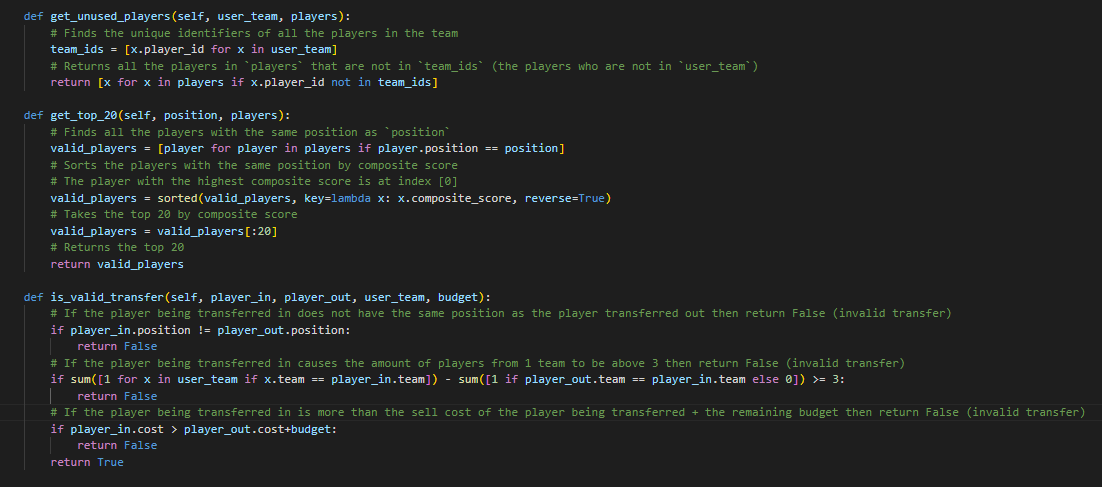
Description automatically generated  
A screenshot of a computer program

Description automatically generated**The final method in `TransferWindow`, is `setup\_list\_view` which is the function used to populate `tranfer\_table` with the suggested transfers and in turn create the layout (which includes `transfer\_table` to populate `main\_widget`. Firstly, it defines and configures `transfer\_table`, setting the row and column counts to 2 and 10 respectively (reasons described above). It then iterates through the 2-D array of transfers converted into an enumerate object, with each transfer being an array in the format [player\_out, player\_in], the players are represented as `base.Player` objects. With every iteration, it places the `.name` attribute of each `base.Player` object in the row of the table indicated by `index`. `transfer\_table` is then added to `layout` which was defined at the start of the method, with layout then being returned. Since this function was called to populate the `main\_widget` of the window, this visualises `transfer\_table` allowing the user to see the recommended transfers tailored to their team.  **ExitWindow:**The `ExitWindow` class represents the window of the GUI application that allows the user to exit the program. It has the functionality to kill the program, or let the user cancel the action and return to the application. It has a very simple initialisation, only calling upon the constructor of the parent class `WindowParent`, settings its own `window\_id`, calling its own `setup\_ui` and calling `apply\_colours` to add the colour scheme to its widgets.   
A screen shot of a computer program

Description automatically generated  
All the functionality of `ExitWindow` is in `setup\_ui` as the operations that it performs are very simple in nature. After configuring both of the buttons present in the window, it connects `exit\_button` and `cancel\_button` to `sys.exit()` and `self.close` respectively; meaning if `exit\_button` is clicked then the program is killed and if `cancel\_button` is clicked then the user returns to the window that `ExitWindow` was opened from (either `MainWindow` or `LoginWindow`).   
 **base.py:**   
**Dataclasses:**`base.py` contains three main dataclasses: `Player`, `User` and `League`. This is due to these classes having no associated methods, therefore, the `@dataclass` decorator is used to improve readability and reduce verboseness. It does this because it automates certain methods such as initialization.   
In each dataclass, on the left of the `: ` is the name of the attribute and on the right of it is a type hint to increase type safety. Additionally, `=` is used to set a default value as in some cases this information will not be required to be updated or accurate.   
**A screen shot of a computer program

Description automatically generated  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
Optimiser Base Class:**The Optimiser base class acts as a storage for a set of attributes and values that both `TransferOptimiser` and `ChipOptimiser` will use in the process of providing their recommendations. The values are as follows:   
- **`lineup\_size` and `bench\_size`:** These attributes set the size of the lineup (11) and the size of the bench (4) for a FPL team.  
- **`team\_max`:** This attribute represents the maximum number of players that can be from one team in an FPL lineup.   
- **`formation\_constraints`:** This attribute provides the minimum and maximum number of players in each position, every FPL team – however the starting lineup must still only have the 11 players limited by `lineup\_size`. Each dictionary key corresponds to a position, and each value is a tuple where the first value is the minimum, and the second value is the maximum.   
- **`position\_constraints`:** This attribute provides the number of players required in each position in the entire 15 man squad, with each dictionary key corresponding to a player position in FPL and each value corresponding to the number of players required.   
**A screenshot of a computer program

Description automatically generated  
TransferOptimiser:**The `TransferOptimiser` class is a child of `Optimiser` that utilizes the values that is stores to assess the available pool of players for transfers and then recommend a series of transfers that the user can make (based on which valid transfers will give them the highest gain in composite score). This function does not use linear optimization (unlike `ChipOptmiser`) as it is not required here due to us only needing to assess a smaller group of higher performing players. It has multiple different methods as follows:   
 **generate\_transfers:   
A screen shot of a computer program

Description automatically generated**`generate\_transfers` takes three parameters in order to generate transfers for the user’s team:   
- **`user\_team`:** This is an array of 15 player objects representing the players that the user has selected for their current FPL team.   
- **`players`:** An array of player objects representing every player in the FPL database.   
- **`budget`:** The balance that the user has for spending on players in their FPL team.   
To start, an array `transfers` is defined, which is left empty for now as transfers will be appended to it later. Then the players that are already in the user’s team must be taken out of the pool of available players – this is done by calling `get\_unused\_players`, another function in `TransferOptimiser`. From this pool of available players that has been returned by `get\_unused \_players`, the top 20 players in each position by composite score are found by using the function `get\_top\_20`, another function in `TransferOptimiser`, with the inputs being the desired position and the pool of available players.   
Using the data that has been retrieved the function can then iterate through the players in the user’s team, comparing them against each player in the user’s team against the top 20 players in their position – if the transfer is a valid transfer, which is found using the function `is\_valid\_transfer`, then it is appended in the format `[player out, player in]` to the array `transfers`. Once this has done for every player in the user’s team, the function returns the transfers array, sorted in reverse order based on:   
*player\_in.composite\_score – player\_out.composite\_score*However, it does so in reverse order so that the transfers that the user will gain the most composite score from are at the beginning of the array rather than the end.   
  
**  
ChipOptimiser:**The `ChipOptimiser` class is a child of `Optimiser` that utilizes the values that it stores to assess the available pool of players and recommending a lineup for the user to choose if they wish to play their wildcard or free hit (1.1.2.3).   
A screen shot of a computer

Description automatically generated   
The class variable `chip\_switcher` should allow for different functions to judge based on different metrics – ideally judging on either composite score or composite score over the next 3 matches. It contains strings as its keys, mapped to lambda functions that allow a given function to use one or the other. However, in this iteration of the application, the ability to judge a player over the next 3 games has not been implemented, so although these functions have been made so they are ready to judge such data (to ease the future development of such a functionality) they cannot do so as of now.   
A screen shot of a computer screen

Description automatically generated  
The first function in `ChipOptimiser` is the `captain\_chooser` function, which takes the users team in the format of an array of `Player` objects as it’s input. It then returns a dictionary mapping the player with the highest composite score to the key `captain` and the player with the second highest composite score `vice\_captain`. It does so with the following lines:   
- **Captain:** max(team, key=lambda player: player.composite\_score)   
- **Vice Captain:** sorted(team, key=lambda player: player.composite\_score, reverse=True)[1]   
`captain\_chooser` has the decorator `@classmethod` so that it can be used without an instance of the class being created, allowing for the function to be used dynamically and with any user’s team.   
A screen shot of a computer

Description automatically generated  
The second function in `ChipOptimiser` is `bench\_boost\_predict`, which once again takes the user’s team as its input. It then returns the combined composite score of the user’s bench – doing so using the sum function and list comprehension: `sum([player.composite\_score for player in team[11:]])`   
The next set of function are all used in order to recommend lineups for the free hit chip:   
**interpolate\_budget:  
A screen shot of a computer code

Description automatically generated**The `interpolate\_budget` function takes the user-specified `bench\_importance` and the user budget as its two parameters and uses them to return a value for the maximum amount of budget that should be used on the starting eleven of the generated team (with the remaining amount from however much of it is used to be spent on the bench).   
First the function checks that the value for bench importance is between 0 and 1, and raises an exception if it is not, as this would cause recommendations that would not comply with the FPL team selection rules. The values `min\_multi` and `max\_multi` are two float values, named for the effect that they have on the amount of budget to be used on the bench. The final value for how much budget is to be used on the bench is found using the following line:   
`return budget\*(min\_multi+(max\_multi-min\_multi)\*bench\_importance)` **get\_position\_requirements:  
A screen shot of a computer program

Description automatically generated**This function is a function to be used when generating the bench, after the starting lineup has been generated. It essentially takes the class’s position requirements dictionary, counting the number of players in each position in the generated starting 11 and then subtracting it from the original position requirements. This essentially finds the remaining positions that the bench needs to accommodate for. It does so in the form of a dictionary and using the `sum()` function and list comprehension.  **generate\_starters:**The `generate\_starters` function is used by the `generate\_team` function to generate the starting 11 for the team. It takes multiple inputs, one being an array of the entire pool of available players in the FPL database, represented by `Player` objects. It also considers the available budget that the user has at their disposal in the form of a float, whilst also taking the `chip\_type` into account (although in the current iteration of the application this is not really utilized as it is only used for the free hit and not the wildcard – see 1.1.2.3). Firstly, the metric by which players are judged by is decided by using the `chip\_type` input and passing it into the `chip\_switcher` dictionary, to return a lambda function which will be used as the metric to judge the players by.   
`generate\_starters` uses the `pulp` python library, to create and solve linear optimization problems with ease, and without having to write complex mathematical equations in python. After defining the metric, a `pulp.LpProblem` instance is created, in the interest of maximizing the given metric. The list of selected players is then created, which after the problem is solved, will be every player marked with a 1 by the algorithm. `pulp` allows for certain constraints to be provided to the problem and the constraints (in readable form are provided as follows):   
- The amount of players selected must be equal to `cls.lineup\_size` (or 11)   
- The combined cost of the player must be less than or equal to the budget   
- For every position, the number of players in each position must be less than or equal to the maximum number of players for that position and greater than or equal to the minimum number of players in that position (as dicatated by `cls.formation\_constraints`)   
- For every given team in FPL, the amount of players from said team must be less than or equal to `cls.team\_max` (or 3).   
The problem is then solved in the interest of maximizing composite score and abiding by the specified constraints using `problem.solve()`. The selected team is then found by iterating through the array of players, and if their associated `pulp.value` is equal to 1, also using list comprehension to do so. The budget used is then by iterating through the selected team and finding the sum of their costs – this can be used to generate the bench, as the remaining budget is to be spent on the bench.   
The final return of the function is: [The starting 11 sorted by position, the budget used].   
**A screen shot of a computer code

Description automatically generated  
generate bench:**The `generate bench` function essentially has the same function as the `generate\_starters` function, except it does not need to adhere to any formation constraints and instead of the returned players having to adhere to the lineup size (11), they need to adhere to the bench size (4). The position requirements are also different, as they are generated using the starting 11 and the `get\_position\_requirements` function. Other than these key differences, it functions the same.   
**A screen shot of a computer code

Description automatically generated  
generate\_team:   
A screen shot of a computer program

Description automatically generated**`generate\_team` is effectively the function that brings all of the other functions together, generating a full starting eleven and bench for the user to view. It takes multiple inputs, one being an array of the available pool of players, which are to be represented by `Player` objects. Another is the budget that the user has in their balance, with another being the specified bench importance. Finally, the `chip\_type` can also be specified for its operation, however as discussed previously, this is not used in this iteration of the application, so it does not really need to be discussed – but the functionality to implement this feature is there already.   
First, the budget for the starting 11 is generated by using `interpolate\_budget`, allowing for `generate\_starters` to be called which then returns the starting 11 and the budget that has been allocated for the bench. The bench budget is then decided by the initial budget subtracted by the budget used on the starting 11 – the positions that the bench is required to fill are also found here by calling the `get\_position\_requirements` function with the starting 11 as its argument. Finally, the bench is generated using the bench budget, positions left and starting 11. The starting 11 and the bench are then returned by `generate\_team` as a singular array in the line `return result\_11+result\_bench`.   
Because `generate\_team` has the decorator `@classmethod` it is able to be called outside of the class, without an instance of the class being present – however it can still use all of the functions inside of the class. This allows for the dynamic generation of suggestions and enhances readability of the code by reducing the boiler plate involved with creating instances of classes.  **RatingSystem:  
A black screen with colorful text

Description automatically generated**`RatingSystem` is a class containing the weights and formula for generating the composite scores of players. The class itself is never initialized as instead, the `@classmethod` decorator is utilized so that local variables within the class can be used without having to declare an instance of the class. Instead, it can be called as follows: `RatingSystem.get\_player\_rating(ppg, total\_points, ownership\_percentage)`.   
Inside the class is a dictionary `player\_weights`, in which the keys are specific attributes that `Player` objects in the system have and the values being their respective rates when creating a composite score. The actual calculation of the composite score is done in `get\_player\_rating`, which has the decorator `@classmethod` so that it can be called without creating an instance of the `RatingSystem` class. It operates by multiplying each attribute of a player by their respective weighting in `player\_weights`, and then adding these together to create a composite score. This composite score is then returned.  **fpl.py:**`fpl.py` is a python file that contains most of the program’s interaction with the FPL API and the application’s own database. It contains the class `FPL` which is initiated when the user logs in to the program.   
The class takes two parameters in its constructor, `session`, which is meant to be a `requests.Session()` object, which in the real program will be provided by `LoginWindow`, the `user` parameter is meant to be a `base.User` object (as this would provide the necessary cookies to access FPL data on an authenticated level).   
**A screen shot of a computer screen

Description automatically generated**  
The first method in `FPL` is a function that returns an array of all the players in the FPL database in the form of `base.Player` objects. These objects are complete with a value for composite score so that this array of players can be used to generate lineups and recommend transfers.   
A screen shot of a computer code

Description automatically generated  
To start, a cursor to the database is opened from the object’s attribute connection, it then executes the SQL `SELECT \* FROM players` and stores the data found from this query into the local variable `data`. In this `data` variable, there will be a series of tuples, each representing a row in the database, therefore meaning that each tuple in the array stored in `data` represents a player in the FPL database. The cursor is closed, and then in the return statement of the function, list comprehension is used to iterate through `data` and turn every tuple into a `base.Player` object. This means that an array of all the players in FPL as `base.Player` objects is returned by this function.   
The next method in `FPL` is `get\_current\_user\_picks`, which is very important to the functionality of the application, as without this the user would not be able to view their team in `LineupWindow`. It is a function that takes no inputs, it simply uses the attribute that the class already has, to return the current users team.   
A computer screen with text

Description automatically generated  
Firstly, the program checks if there is a user logged in, if there is not then it returns False, but if there is then the program may continue (as if the user is not logged in, we can’t find their lineup). If there is in fact a user logged in to the application, `utils.fetch` is utilized to retrieve the data from the `current team` page of the FPL API, using the user’s cookies to authenticate. This returns a JSON string which is stored in `data`, of which the data that we need is in the key [‘picks’], therefore the array `ids\_list` is created and list comprehension is used to retrieve the ids of every player in `data[`picks`]`. This list of player ids is then returned, after which it will be interpreted and formatted correctly by other methods.   
The next few methods in `fpl.py` are simple functions that use `utils.fetch` to make calls to the FPL API, to retrieve data on various user and league related items. They are used throughout the program with `get\_current\_user\_balance` being used to great effect in order to generate transfers and lineups.  
**A screen shot of a computer code

Description automatically generated  
  
  
  
utils.py:**   
**URLs:  
A computer screen with text

Description automatically generated**The constant `base\_url` represents the common base URL for the Fantasy Premier League API, with the `api\_urls` dictionary serving as a collection of various specific endpoints, each identified by a key string relating to its functionality and purpose. This method of representing the URLs used for API calls allows for dynamic generation of different URLs, as any URL with `{{}}` means that relevant player, league and user ids (or any similar ids) can be substituted when making actual API requests in the relevant methods. Representing the URLs in such a way also allows for the centralized management of API endpoints, meaning it is easier to locate and update URLs as required. This centralized approach enhances the maintainability and readability of the code, for instance if I wanted to update a specific URL in the application, I would not have to change every instance of the URL in my code, I could just change this one dictionary.   
**Write/Read JSON:**These two methods, `write\_json` and `read\_json`, ­are used for the handling of the `settings.json` file, in tandem with the `Settings` object of every window in the application.   
`write\_json` takes a JSON string and a designated path as its parameters, then opening the specified path in ‘w’ mode, so that it can write to the designated settings file. It does so with the following line `json.dump(data, f, indent=4)`, which serializes the data and writes it to the file; with an indent of 4 to ensure that it is more readable from a human perspective in case any debugging is required.   
`read\_json` takes the path of the JSON file, opens it in ‘r’ mode so that the contents can be read and then returns it using the following line: `return json.load(f)`. This is all that is required in terms of formatting in this method, as the data will be correctly formatted elsewhere – in a `Settings` object for instance.   
These methods may seem void of any proper error handling at face value, however this problem is addressed during the actual usage of the function, as different error cases require different reactions from the application. Therefore, most of the error handling is performed during the actual usage of the methods – in a `Settings` object for instance.   
A screen shot of a computer program

Description automatically generated  
 **Fetch:  
A computer screen with text and images

Description automatically generated**This function is used for the interaction with the FPL API at a fundamental level, with this function returning the JSON data found at a given API endpoint. The `fetch` function takes 5 parameters:   
- **`session`:** The function must be provided with a valid session in order to interact with the FPL API, this will usually be an instance of `requests.Session()` provided by the an FPL object.   
- **`url`:** This represents the URL with which with HTTP GET request will be made to, URLs will be formatted using `base\_url` and `api\_urls`.   
- **`cookies`:** Certain API calls will require user authentication (such as viewing a certain users team), therefore the cookies of said user (the ones that show to the API that we should be provided access) must be provided. Here they are provided in the form of a dictionary. If a cookie is not passed as an argument when calling this function, then the value of `cookies` will default to None, as not all HTTP GET requests that can be made require user authentication.   
- **`retries`:** This parameter dictates the number of times to retry the request in case of failure, it has been left as a default value (2) so that it can be changed easily if the development of the application necessitates it.   
- **`cooldown`:** This represents the time in seconds that are to be left between each retry. Once again it has a default value so that it can be changed if required later on in development, the default is set to 1 second.   
Firstly, the function initiates a loop that iterates until a successful response has been generated or the maximum number of retries has been reached. In said loop, the `session.get` method is used in order to make a request to the specified URL with the provided cookies. If this generates a response, it is then parsed as JSON using `response.json()`, so that it can be used in our application, and returned – ending the loop. However, if an exception occurs the function increments `retries\_count` (which starts at 0) by 1. If the specified `retries` limit is reached, then the program will raise an exception to indicate that there has been too many retries. For the cooldown between retries, `time.sleep(cooldown)` is used, to provide a slight pause in the runtime of the program.   
A computer screen shot of a program

Description automatically generated **Account Cookies:**  
  
This function is used to interact the Fantasy Premier League website (fantasy.premierleague.com) in order to log in using the user’s credentials, and then return the cookies used for authentication on the website. Similar to `fetch` it takes a variety of parameters:   
- **`session`:** Once again, the function must be provided with a valid session to operate, however unlike `fetch`, the `requests.Session()` object that will be used in this function should be provided from an instance of `LoginWindow` as that window is the only point in the program where user credentials should need to be obtained.   
- **`email` and `password`:** Since both a username and password are both required for login authentication on the Fantasy Premier League website, they must both be passed into the function as arguments so that they can be used in the request.   
- **`retries`:**  This parameter dictates the number of times to retry the request in case of failure, it has been left as a default value (2) so that it can be changed easily if the development of the application necessitates it.   
- **`cooldown`:** This represents the time in seconds that are to be left between each retry. Once again it has a default value so that it can be changed if required later on in development, the default is set to 1 second.  
Firstly, the function initiates a loop that should continue until a successful login operation is performed, or until the maximum number of retries has been reached (as dictated by `retries`). Unlike in `fetch`, since we are sending data to the website and not requesting it, `session.post` is used in order to POST a request to the login endpoint (`api\_urls[‘login’]`), with the provided email and password also included in the POST. Other items are also included such as `app`, `redirect\_uri` and certain headers are also included, this is simply so that it is in the right format for the FPL website. If this POST request returns a valid response, then the data is not processed any further and instead, a dictionary containing the specific cookies `pl\_profile` and `datadome` is returned. However, if an exception occurs during the login attempt, `retries\_count` (which starts at 0) is incremented, if it reaches the specified limit `retries`, then `False` is returned. For the cooldown, between retries `time.sleep(cooldown)` is called, to provide a brief pause in the application’s runtime.   
**connect\_to\_db:**This function is used to create a connection object to the SQL database using the specified parameters:   
- **`host`:** The IP address of the MySQL database server.  
- **`user`:** The username used to authenticate to the server.   
- **`password`:** The password used to authenticate to the server.   
- **`database`:** The name of the database that needs to be accessed.   
This function allows for the connection to the database to be defined just once (in the case of this application in the FPL object being used), and then accessed dynamically throughout the application – this removes the need to open and close connections to the database throughout the program, taking less toll on the resources of the computer.   
A computer screen shot of a program code

Description automatically generated  
**get\_current\_gameweek:  
A screen shot of a computer program

Description automatically generated**The function `get\_current\_gameweek` uses `fetch` to access the static endpoint of the FPL API which returns a JSON string to be stored in `data`. It then selects the list of gameweeks by using the key `[‘events’]` and iterates through the array of gameweeks checking if it has the property `[‘is\_current’]=True`, and when it does it returns the `id` of the same gameweek.   
 **convert\_team:**  
In the FPL API, each team in the Premier League has an associated integer value, for instance Arsenal is 1. In the player data, their team will not be ‘Arsenal’ it will be ‘1’; this means that there has to be some way to convert from the integer format to the string format so that the user can understand what team a given player plays for. In my application, this is done by mapping each team name to their respective integer value in a dictionary, in a function where the input is the team’s id. Due to this dictionary mapping, the return of the function can simply be the dictionary with the key being the team id entered.   
A screenshot of a computer screen

Description automatically generated  
  
  
  
  
  
  
  
  
  
  
  
  
 **convert\_position:  
A computer screen shot of a program code

Description automatically generated**Similar to real-life teams in FPL corresponding with certain integer values, positions also have corresponding integer values that need to be converted between in order for the user to understand their significance. Once again, the different positions are represented by string and mapped to their respective integer values in a dictionary. So, again the return can simply be the dictionary with the key of the position id entered.   
 **update\_player\_table:  
A black background with colorful text

Description automatically generated**In order for transfer recommendations and lineup recommendations to be accurate and helpful to the user, the data and information that the application has on players must remain current and up to date. Therefore, the user must be able to choose to update the player table so that they can get accurate transfer recommendations – player data would be updated automatically on my own server but unfortunately, I do not have access to one.   
Firstly, a database cursor is initiated from the `connection` parameter that in practice will come from an instance of the `FPL` class. To update player data, the `fetch` (using the parameter `session`) function is used to retrieve player data from the static endpoint of the API, with the player data being contained in an array `[‘elements’]`. For each player in the array, an SQL statement using the relevant player data INSERTs the new values into the table and if there is a duplicate player it updates the table with the relevant player information instead. After the iteration through the array is finished, the changes are committed to the database and the cursor is closed.   
**update\_composite\_score:   
A black background with colorful text

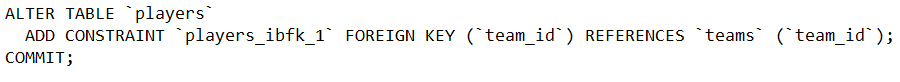
Description automatically generated**Whilst it is entirely possible for us to update most rudimentary player statistics using `update\_player\_table`, in order to update the composite score of each player, we must use the data that is already in the table.   
The function `update\_composite\_score` takes 1 argument `connection` which should be taken from the database connection of an `FPL` object. SQL is used to fetch an array of all the players (with the data they store being represented in tuples), with the array then being iterated through and an SQL command being created to update the composite score of each player, based on the data that we currently have on each player. `base.RatingSystem.get\_player\_rating` is used, with the inputs being: total\_points, points\_per\_game and selected\_by\_percent. The changes are then committed to the database and the cursor is closed. **create\_player\_object:  
A screen shot of a computer

Description automatically generated**For the application to be able to process and store player data, it must be in the form of the `Player` class that was defined in `base.py`. Therefore, there must be some way to create player objects dynamically in the application. The `Player` objects are initialized from the data in the `players` table of the database.   
The input of this function is the player id of the desired player. First, a cursor is created and all the players in the database whose id is equal to the input id are fetched, with the return being an array of tuples with each tuple containing the data that is stored on a player. Since there is only one tuple in the array – because the player id is a unique identifier – the player object can be created with the following line `return base.Player(\*data[0])`, as the first element in the array is the only element in the array.   
**create\_user\_object:   
A computer screen shot of text

Description automatically generated**For the application to be able to process and store user data, it must be in the form of the `User` class that was defined in `base.py`. Therefore, there must be some way to create user objects dynamically in the application. The `User` object is initialized from data retrieved from the FPL API.   
The parameters of this function are `session` and `cookies`, with `session` being intended to be a `requests.Session()` object from an instance of `LoginWindow` and `cookies` intended to be a dictionary of `pl\_profile` and `datadome` also from an instance of `LoginWindow` after a successful logon. The process of the function is as follows, the `fetch` function is used with both parameters as the `session` and `cookies` and `api\_urls[‘me’]` as the URL, because we have the cookies to authenticate this request, this returns a JSON string of data on the current user. However, since the `User` object only requires an id, name, email and dictionary of cookies, it can be initialized and in turn returned as follows: `return base.User(data[‘entry’], data[‘first\_name’], data[‘email’], cookies)`.   
**merge\_sort:**`merge\_sort` is a divide and conquer algorithm, it works by recursively diving the given array into two halves, then sorting each half and then merging the two sorted halves. In my application it works as follows:   
Firstly, if the array provided is empty, or has only one element, then the function returns that array as it does not need to be sorted (base case of the algorithm). If this is not the case, then it splits the array into two halves (left half, right half). This continues as `merge\_sort` is called recursively for each left half and right half – sorting each sub array of the given array. The sub arrays are then merged, by comparing elements from the left half and right half appending the smaller (or larger depending on if they key has dictated that it is to be reversed) element to the array `merged`. This process of merging continues until one of the sub arrays has been exhausted completely. After the `merged` array is filled it is returned as the original array sorted.  
I have used it for sorting players by specified metrics in the `DatabaseWindow` as it has a time complexity of O(nlog(n)) making it efficient for sorting the significant number of players in the FPL database. Additionally, its use in tandem with `sort\_switcher` and the lambda functions it contains (which are used as they key for sorting) allows users to dynamically customise sorting based on different attributes. In addition, the separation of the sorting algorithm from the rest of the code in `DatabaseWindow` and the use of lambda functions to judge players by certain attributes helps to maintain a more readable and modifiable codebase for future development amd debugging.  
A computer screen shot of a program code

Description automatically generated  
 **Database:**In my solution, I have used cpanel, as it was provided by my institution, in order to run an SQL server to meet the data needs of my application. The following is the SQL query used to create the `players` table in the database:   
**A screenshot of a computer code

Description automatically generated**  
And the following is the SQL query used to create the `teams` table  
**A computer code with black text

Description automatically generated**  
And in order to link the two tables by ‘team\_id’:   
****  
**A close up of words

Description automatically generated**The players table was initialled populated by using the method `utils.update\_player\_table`.   
**A screenshot of a computer

Description automatically generated**  
Data in the players table as of 25/01/24 (through usage of the application users will be able to update the data present by pressing the button to update player data on `MainWindow`). As seen in the code of the solution, the database was connected to using `mysql.connector`.  **4 – Testing  
4.1 Test Plan:  
4.2 Test Cases:  
4.2.1 LoginWindow:**  
**4.2.2 MainWindow:  
4.2.3 LineupWindow:  
4.2.4 LeagueWindow:  
4.2.5 Database Window:  
4.2.6 SettingsWindow:  
4.2.7: ExitWindow:**  
**5 – Evaluation   
5.1 Measurement of Success:**The success of my solution will be measured on the requirements that were produced in Section 1.3, which were based off surveys and my own personal investigation into the current system (the FPL website). The requirements for my project outlined that the application should be able to:   
- ‘Users must be able to log in to their FPL account using their email and password’: This has been accomplished, as when users enter the correct credentials for their FPL account, they gain access to the application.   
- ‘Users must be able to view their current rank in the ‘Overall’ league in the main window’: This has been accomplished as in MainWindow there is a label that informs the user of their rank in the overall league.   
- ‘Users must be able to view their current FPL team’: This has been accomplished as users are able to view their current FPL team in list view in the LineupWindow.   
- ‘Users must be able to view the optimal lineup for the next gameweek’: This has been accomplished as in the LineupWindow, users can view the optimal lineup for the next gameweek as if it were their own.   
- ‘Users must be able to request recommended transfers (and be supplied with them in turn): This has been accomplished as in the LineupWindow, there is a button that the user can press to open the TransferWindow, which generates 10 transfers tailored to their specific team.   
- ‘Users must be able to access the player database and must be able to sort the database according to a variety of metrics (including composite score)’: This has been achieved as this is possible to do in the DatabaseWindow   
- ‘Users must be able to change the colour scheme, font used’: This has been achieved as users can change these features in the SettingsWindow   
- ‘Users must be able to log out’: This has been achieved as users may log out in the SettingsWindow if they so desire.   
- ‘Users must be able to exit the application without clicking the ‘X’ that is in the top right of the application by default’: This has been achieved as it can be done through the ExitWindow   
  
Since all the requirements have been adequately met, I feel that it is fair to say that that the outcome of the creation of this application has been quite successful. However, throughout the development of this project, there were multiple features that I would have wished to include in this iteration, that I simply did not have the time to include. These partly/not implemented features will be discussed further in the next section.  **5.2 Possible Improvements:**There are numerous possible improvements and features that could be implemented that I found during development, and many of these I would very much like to implement in the future in my own time.  
**Improvement of the player rating system:**An improvement of the player rating system could be very beneficial to the recommendations generated by `ChipOptimiser` and `TransferOptimiser`, as they would provide more accurate and valuable insights into how players compare to each other based on their composite score. An improvement to the player rating system could be introducing more in-depth statistics into the `RatingSystem`. Where these would be accessed from is the only problem as there is not real automate way to link player statistics from a website that focuses on real-life football (such as FBREF) to FPL. This is because the unique identifiers that are given to players in FPL operate only in FPL. This makes automation of assigning a certain players id on the FPL website to their id on another website very hard. Therefore, any improvement of the player rating system, in such a way, would require a vast amount of research into how two instances of a player on two different sites could be linked. However, another way that the player rating system could be improved is by redrawing the weights associated with each statistic, which could require some more surveys and research into the field. This would be a much easier task, but nevertheless I would like to investigate both possibilities in the future in the interest of improving the application as much as possible. **Introduction of in-house team rating system:**An in-house team rating system so that FDA (Fixture Difficulty Analysis) can be done, is another feature that could improve the application greatly. This is because it allows more in-depth composite score generation as it would provide the algorithms with data that is hopefully more accurate and up to date – and related in a stronger fashion to how FPL points are scored. One way this could be implemented is by ranking teams based on how many points FPL points they concede per gameweek. As for instance, the current FDA works based on how two teams compare in a real-life scenario, however in an FPL scenario a ‘worse’ team could very easily have less points scored against them per gameweek than a team that is `better` than them. Therefore, this would be a great feature to implement as it would help to improve the accuracy of the player rating system – which is the crux of all transfer and lineup recommendation.  **Implementation of generating wildcard suggestions:**Whilst in its current state, the `ChipOptimiser` class is capable of handling both free hit and wildcard lineup generation (1.1.2.3), the application is not yet capable of providing the class with the data needed to make decisions based on multiple games – meaning wildcard suggestions cannot be generated. Once again, I would like to implement this feature in the future as it could provide the user with more varied and interesting team selections rather than just the best players for the next gameweek.   
This could be implemented in tandem with the improvement of the player rating system, as instead of generating a score for the next gameweek, fixture difficulty for the next 3 gameweeks could be taken into consideration, as well as how many games that the player could possibly play in those gameweeks. So, this improvement could further improve the player rating system, but also introduce a new feature entirely.  **Implementation of formation view in LineupWindow:**Whilst the ability to switch between the list widget and formation widget has been implemented, the ability to view lineups in formation form has not yet been implemented. I would like to implement this in the future as it could go a long way to enhance the user experience – as some people may prefer to see the players visualized rather than simply names on a sheet.   
A way to implement this could be to have a series of images representing the players in the shape of the formation that has been provided (e.g. 1 goalkeeper, 3 defenders, 5 midfielders, 2 forwards). There could then be labels underneath the images indicating name and cost. However, any implementation of this feature would require further investigation and research to ensure that the functionality provided is as complete as possible.  
In addition to improvements that could be made to the solution, there could also be improvements to the process of development due to lessons learnt. In hindsight, I slightly regret the inclusion of a significant amount of customisability in the user-interface – this is since it took up quite a large amount of development time for something that did not really affect the product in a particularly impactful way. I would now much rather have spent my time developing more features for the application, such as the ones list above, not only because it would have improved the solution even more, but because I found much more enjoyment out of creating features such as transfer recommendation and lineup optimisation when compared to creating the customisability features for the application.   
Another lesson I have learnt regards time management, when taking breaks between development I often took breaks that lasted too long (more than two weeks for instance), meaning when I came back to the project, I often had to relearn the codebase to implement any meaningful features at all. This was ultimately lost time as I could have implemented more and perhaps better features if I managed my time in a better manner.   
Finally, **5.3 Conclusion:**In conclusion, due to the testing completed and feedback taken in, the application has met all its requirements from the research stage. Most of the primary features that could be replicated from the FPL website have been replicated, and certain improvements have been made – such as the option for recommendations and more detailed players statistics being readily available. The application is also poised for future development, perhaps the developments mentioned in 5.2, due to considerations I took during its development, such as the ease of readability and maintainability of the codebase.