Development of Fantasy Football Game

Including Real-Time Auction Component

Stuart Wright

May 9, 2020

BSc Computing Project Report

Birkbeck College, University of London, 2020

Abstract

This is the abstract.
It consists of two paragraphs.

Contents

Introduction

This report documents the development of a multiplayer fantasy football game, implemented as a web application.

Terminology

In this document, **football** means association football, or soccer as it is known in some countries.

The word **player** will always refer to a real-life football player, as opposed to a person playing this fantasy football game. The person using the application will typically be referred to as the **user**, but depending on the context they may also be referred to as a **manager** (of their fantasy football team) or a **participant** (in an auction).

Similarly, a distinction between real-life football clubs and fantasy teams is necessary. A real-life football club (e.g. Liverpool or Arsenal) will be referred to as a **club**, whereas where the word **squad** is used, it will always refer to a fantasy team. The word **team** may refer to either, but its meaning will be made clear from the context.

Fantasy Sports

Fantasy sports are games in which participants build imaginery teams consisting of real sports players. These fantasy teams then compete against each other, scoring points based on the real-life performance of the players in their team.

Requirements

A list of high-level requirements was drafted up for the proposal, split into two sections:

- Requirements for Minimum Viable Product (MVP).
- Additional features to be added as time permitted.

All MVP requirements were successfully implemented, along with some of the additional features.

User Stories

Each item on the initial list of requirements could be considered a user story. A user story is typically a few sentences of simple, non-technical language, which explains the desired outcome from the user's perspective. [1]

During development of this application, user stories were organised on a Trello board. An example of how the board looked during the early stages of development can be seen in figure 1.

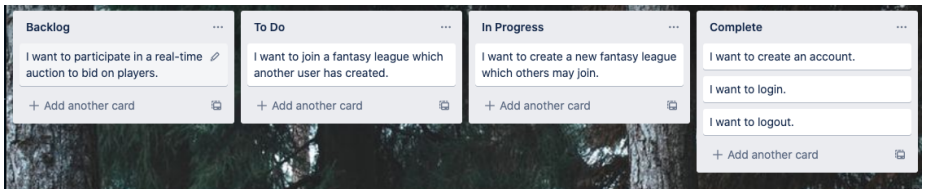

Figure 1: Trello Board

The **In Progress** and **Complete** sections are self-explanatory. The distinction between **Backlog** and **To Do** exists to prioritise certain stories ahead of others, although the intention is still that stories in the **Backlog** list will be completed. In the example shown, it made sense in the context of building this application that users must be able to join leagues before they can participate in the auction. This is why the latter was placed in the **Backlog** list until development of this feature could realistically begin, and only at this point was it moved to the **To Do** list.

Minimum Viable Product

The MVP requirements stated that the user must be able to carry out the following tasks:

- Create an account, login to said account, and logout.
- Create a new fantasy league which other participants may join.
- Join an existing fantasy league.

- Participate in a real-time auction, during which they will bid against other participants in their league on real football players to join their fantasy team.
- After the auction is completed, view records of points scored by their own team, and other teams in their fantasy league, as points are scored based on performance of the football players in real games.

With the exception of the first item on the list above, about which there is little of interest to discuss, some further detail on each high-level requirement follows.

Create a League

The bare minimum requirement here was that the user must be able to create a fantasy league which others could later join, and give it a unique name of their choosing. This is the version which was implemented in early iterations, to allow development to progress as quickly as possible.

In later versions, additional functionality was added to allow the league creation process, giving the user more control over the rules of the game. Some examples can be seen in figure 2.

The complete list of options available to the league creator is:

- **League name**
- **Number of Participants**
- **Event** - this refers to the real-life football fixtures in which points will be scored. For example, 'Premier League Week 1'.
- **Max Players Per Club** - setting a low value here helps to avoid a situation where only players from the best clubs are selected.
- **Number of Goalkeepers/Defenders/Midfielders/Forwards** - these settings to refer to the four main positions for football players. The default setting is 1 goalkeeper, 4 defenders, 4 midfielders and 2 forwards - 11 players in total, which is how a real-life football team might line up. However, if a league creator wants each manager to build a bigger squad of players, they can change this setting. Alternatively, they might want a very fast auction, in which case they could limit the total squad size to only 5 players. There is no reason that a fantasy team's composition must match that of a real football team.

Join a League

Users must be able to join leagues created by other users, subject to constraints:

- A user cannot join a league they have already joined.
- A user can only join a league if the auction has not yet started, and the league is not full.

Before they can join a league however, the user must be able to view a list of available leagues which they are permitted to join. An example of this can be

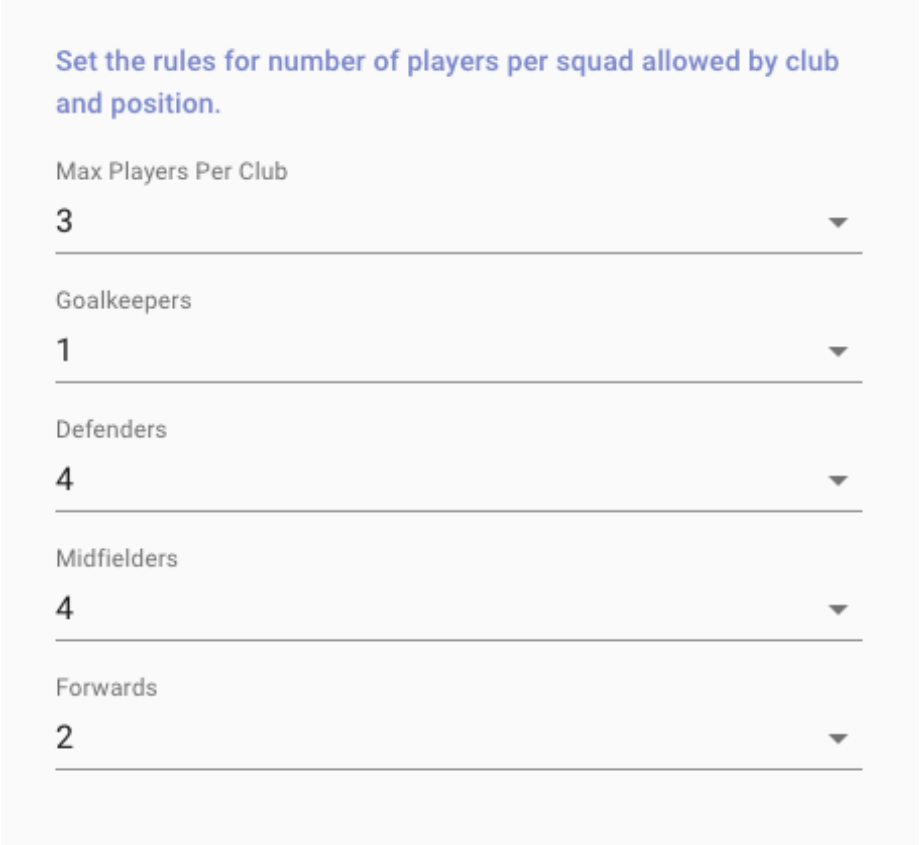

Figure 2: League Creation Screen

seen in figure 3.

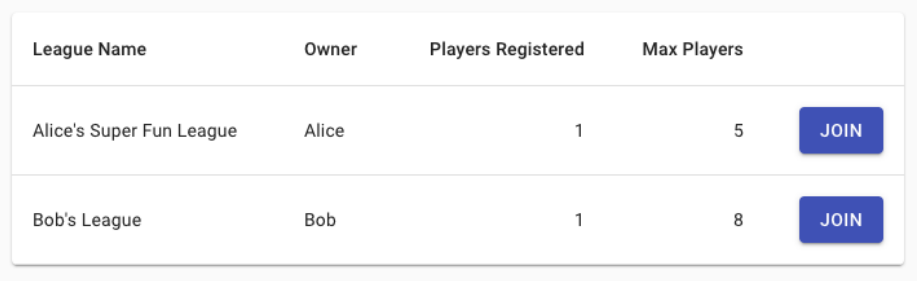

Figure 3: Available Leagues Screen

Participate in Auction

The requirements for the real-time auction component of the application had to be fleshed out significantly before development could begin. The following more detailed rules were drawn up to describe how the auction logic was expected to function:

- The auction can only begin once the league is full.
- Once the league is full, it is up to the league creator to trigger the start of the auction.
- All auction participants start with a budget of £100M, with which to purchase players.
- Once the auction has started, participants must take it in turns to pick a player to be auctioned off to the highest bidder.
  - The participant which selected a player is considered to have opened the bidding at £0.
- All bidding must occur in real-time, with details of bids shared immediately with other participants.
- A player is sold to the highest bidder after 10 seconds of no bidding.
- A participant may be prevented from bidding on a certain player for any of the following reasons:
  - They do not have sufficient budget to make a bid which is higher than the current highest bid.
  - They are already the highest bidder (you cannot try to outbid yourself).
  - They have too many players from the club in question (e.g. Liverpool).
  - They have too many players in the position in question (e.g. defender).
- The auction can only end once all participants have completed a full squad of players as defined by the rules.
  - No participant can ever be left with an incomplete squad. Even if they spend their entire budget on the first player, in the worst case scenario they can still pick up players for free at the end once everyone

else has completed their squad.

Additional Features

The list of optional additional features in the proposal stated that ideally, the user would be able to:

- Participate in more than one fantasy league at a time.
- Make changes to their team after the initial auction.
- Set up automatic bidding by preselecting the maximum amount they would be willing to bid on each player.
- Access the web application using a mobile-friendly user interface (responsive design).
- Customise league with different options relating to the rules of the game.

Additional Features Implemented

The requirement to allow the user participate in more than one fantasy league at a time was implemented as intended. Although it would be impractical for a user to attempt to participate in multiple auctions at once (given the real-time nature of the game), it is technically possible - all they would need to do is open multiple browser tabs and browse to the appropriate league in each. A more practical application of this feature might be to allow a user to register or create some leagues for next week's fixtures, while they are monitoring points being scored for their existing league.

The ability to customise the league with different options has already been covered as part of the **Join a League** section above.

Additional Features Not Implemented

The application does not currently allow the user to make changes to their squad after the initial auction, nor does it allow the user to set up automatic bids. Both are features which could potentially improve the game, but there is also a concern that they could detract in some way from the auction, for the following reasons: * If a user can edit their squad after the auction, the auction holds less importance. * The idea of automation is perhaps more suitable for a business application such as on online shopping site, rather than a fun game in which the auction is supposed to be one of the most enjoyable parts.

Some attempt was made to make the user interface responsive, but ultimately the amount of information that is necessary for the user to see while playing a real-time game such as this made it difficult. In its current state, the game is not fit for consuption on a mobile device, but works well on anything bigger than larger tablet (e.g. 10.2" iPad). Although difficult, this is a feature that would surely have to be implemented for the game to have appeal in 2020.

Design

Some elements of the design process were carried out prior to development beginning. These included:

- High level architecture - identifying the different components required to form a complete web application, and how they will communicate with each other.
- Data modelling - identifying which entity classes were required, which fields they would contain, and their relationships to each other.

Other elements were left until after the development process was already underway. For example, the first sketches of the user interface for the auction were not drawn until after a large part of the server side development had been completed.

High Level Architecture

The first choice was to decide which of the following two approaches to take:

- The traditional approach, which involves most of the application logic being performed on the server, with appropriate HTML returned to be displayed in the browser.
- The 'Single Page Application' (SPA) approach, which allows for most of the user interface logic to be handled by client-side code which simply consumes data from the server.

An article on Microsoft's website[2] states that the traditional approach is better suited to websites with simple client-side requirements, and that SPAs are better suited to applications which require more complex user interface functionalities than what basic HTML forms can offer. Given the requirements of this application, the SPA approach was selected.

This decision meant that the server would be responsible simply for providing the client with the appropriate data, rather than returning HTML pages to display. Although the specific needs of this application meant that real-time bi-direction communication was a requirement, it was also necessary to consider more traditional requests. For example, a user logging in, or requesting a list of available leagues to join. For this reason, it was decided to implement a REST API on the backend in addition to whichever means of facilitating real-time bi-directional communication was chosen.

API stands for Application Programming Interface - a set of rules which allow programs to talk to each other. REST stands for Representational State Transfer. A REST (or RESTful) API is an API which follows a particular set of rules - it receives requests from client programs, and sends a response. There are different types of request for different purposes. A GET request typically involves the client program making a request for data from the server. In this application for example, that might be a user requesting to see a list of available leagues. A POST request will typically involve the client sending some new data to be

written to the server's database. In this application, a user creating a new league would be an example of something for which a POST request would be appropriate.

Regardless of the request type, a REST API typically sends a response containing some form of data. There are a number of different data formats which can be chosen, but JSON (JavaScript Object Notation) is by far the most popular data format for exchange of information in web applications, so this is what was selected.

A diagram showing this high-level architecture can be seen in figure XYZ.

Data Modelling

The following entity classes were identified as necessary in order to build an application which fulfilled the requirements:

- **User** - A user of the application.
- **Player** - A real-life football player.
- **League** - A fantasy league created by a user.
- **Event** - A round of fixtures in real-life football.
- **Auction** - A separate entity for the league's auction, to prevent **League** from becoming excessively complex.

While designing the **Auction** class, it became clear that it would need to be composed of other smaller entities:

- **AuctionUser** - One particular user in the auction.
- **LiveAuctionItem** - The auction item (a player) which users can currently bid on.
- **SoldAuctionItem** - A sold auction item (a player) which has been auctioned off.
- **Bid** - A bid on an auction item.
- **SquadItem** - A player which has been added to an auction user's squad.

Finally, two other entities were added quite late in the development process, as they were not considered during the design phase. They are included here for completeness:

- **PostAuctionUser** - For any data relating to a user in a league after the auction is completed (e.g. how many points they have scored).
- **FinalSquadItem** - Similar to **SquadItem** above, but amended for the post-auction phase of the game (again, the abilty to score points was an important factor).

An entity-relationship diagram modelling the relationships between these entities can be seen in figure ZZZ.

Implementation

Technology Stack

The MERN Stack was chosen to develop this application:

- **M**ongoDB - a NoSQL database management system.
- **E**xpress - a Node.js web application framework.
- **R**eact - a JavaScript library for building user interfaces.
- **N**ode.js - a JavaScript runtime capable of executing JavaScript on a server.

In addition, the Socket.IO library was selected in order to facilitate the required real-time bi-directional communication between client and server.

Part of the motivation for choosing this stack was its popularity. There are several benefits to choosing a popular techonology stack:

- If a stack has become popular, it is likely that the technologies work well together.
- If any problems are encountered, it is likely that somebody else has encountered the same problem before.
- Libraries and documentation are likely to be regularly maintained.

That said, some further research was conducted to ensure that this was the right stack for this particular application, and the findings follow.

React

The author was already comfortable with React prior to beginning this project, and was satisfied that it would fulfil the requirements. Therefore, no alternatives were researched. React user interfaces are composed of components which are updated when the data changes, which is exactly what was needed here. For example, when a new bid is made during the auction, the entire page should not update, but only those elements which are relevant.

React is a library, as opposed to a fully-fledged framework (such as Angular). It does not make assumptions about the rest of the technology stack[3]. This was particularly attractive in this case, as there were unlikely to be any problems integrating whichever libraries were required for real-time bi-directional communication.

Node.js

The most obvious benefit to choosing Node.js for the backend is the convenience factor of writing the same language for server-side code as used for client-side code. Switching between languages involves some cognitive overhead on the part of the developer, and avoiding this should lead to a more efficient development process.

Using Node.js in web applications also opens up the possibiliy for code re-use across different parts of the application. For example, in this application it seemed likely that both client and server side code might have to perform a function such as filtering a list of players down to only those which haven't been auctioned off yet.

In the previous section, JSON (JavaScript Object Notation) was identified as the format for data transfer. This makes Node.js a particularly convenient choice - as the name suggests, JSON can easily be converted to JavaScript objects (and vice versa). The developer can spend less time worrying about the appropriate data structure to represent the data, and more time thinking about how to implement the business logic.

The above reasons made choosing Node.js attractive from a developer experience standpoint, but most importantly, research also showed that Node.js was a popular choice for applications which require constantly updated data such as chat rooms and games. Requests are processed asynchronously without blocking the thread, which means that it is capable of short response times, a necessity for this application.

The main drawback to choosing Node.js seemed to be that it could experience performance bottlenecks for computationally heavy tasks, but this was not a concern for this application.[4]

Socket.IO

Upon learning about Socket.IO, it was clear that this was exactly the library for use in this application. It offers support for event-driven real-time bi-directional communication between the client and server, and abstracts away the underlying complexity of implementing WebSockets. This seemed like a good choice, as implementing the appropriate business logic would be complex enough without also worrying about the low-level details.

Express

Express is the most popular web framework which runs on Node.js, and it is featured in an example in the Socket.IO documentation.[5] With support for Socket.IO integretation and the ability to rapidly develop REST APIs, there was little need to explore alternatives to Express.

MongoDB

While this application could have been successfully developed with a traditional relational database instead, MongoDB seemed like the more appropriate choice for two reasons.

Firstly, MongoDB is particularly convenient to work with in JavaScript applications. Objects stored in a MongoDB collection are very similar to plain

JavaScript Objects in their structure, thus there is no impedence mismatch when representing data from the database in the application.

Secondly, nested data structures seemed more appropriate than tables for the entities required for this application. For example, the idea of an auction containing an array of auction users, and each auction user containing an array of players they've won, made more sense conceptually than having these entities spread across different tables in a relational database.

Development Process

The development process involved taking one user story at a time, and implementing all that was required to make some minimal functional version of that user story a reality. This would typically involve working with the database, backend application logic and the user interface. At this point, some testing was carried out to ensure the feature was working as intended, and usually a few more similar cycles would follow before the feature could be considered to be working as intended.

Work was completed in approximately the following order:

- Login system for users to create account, log in and log out.
- Funtionality to allow users to create and join leagues.
- The auction.
- The post-auction section.

For each feature, work was typically done on the backend first so that each endpoint was returning to correct data for each type of request it might receive. This made development of the frontend significantly easier.

Some user stories required significantly more work than others. Building a basic login system was relatively straightforward, but building a real-time auction was not. As a result, the more complex user stories would be broken into smaller more manageable chunks - for example, first adding the functionality to allow a user to select a player to be auctioned off, and then only once this feature was working as intended, then beginning work to allow others to make counter bids.

Development Tools

Git/Github

Git was used for version control. Typically, code was committed to the repository once any milestone was reached. Sometimes these milestones would relate to a user story, but other times they would relate to a bug fix or some refactoring.

Github offered a centralised storage solution for the Git repository, which made it easy to work on this project on different machines. This was particularly useful when it was time to deploy the application to a production server.

Visual Studio Code

Postman

Postman is a tool which can be used to test REST API endpoints. For example, once the server side code had been written to allow the user to create a league, Postman was used to send a request to the server in the same format as the one that the client would send. It could then be verified that the league was created correctly in the database, and the correct response sent to the client, before work would begin on the client side code.

Yarn

Yarn is a package manager for Node.js projects. Most Node.js projects, including this one, involve the use of several libraries. Yarn helps the developer to keep track of which depencies are required, so that when the code is deployed on a new machine, the process of installing these depencies is automated. NPM (Node Package Manager) is very similar and would have been a suitable alternative here.

Babel

Babel is a JavaScript transpiler, which allows the developer to write code using the latest JavaScript features without worrying about compatibility issues. Babel will transpile modern JavaScript into a backwards compatible version of JavaScript. This significantly improves the development experience, as several useful features have been added to JavaScript in recent years.

Backend

The server side program consists of a REST API

Frontend

Testing

User Acceptance Testing

Edge Cases

Summary

Reflection

Future Improvements

Bibliography

1. Rehkopf, M. (2020). User Stories with Examples and Template. Available at: https://www.atlassian.com/agile/project-management/user-stories/.

2. Microsoft (2019). Choose Between Traditional Web Apps and Single Page Apps (SPAs). Available at: https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/choose-between-traditional-web-and-single-page-apps/.

3. React (2020). React Home Page. Available at: https://reactjs.org/.

4. Altexsoft (2019). The Good and the Bad of Node.js Web App Development. Available at: https://www.altexsoft.com/blog/engineering/the-good-and-the-bad-of-node-js-web-app-development/.

5. Socket.IO (2020). Socket.IO Documentation. Available at: https://socket.io/docs/#Using-with-Express.