Development of Fantasy Football Game

Including Real-Time Auction Component

Stuart Wright

May 16, 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, featuring a real-time auction, 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 selected. Fantasy sports predate the Internet[1], but the game format is well-suited to online play.

There are several variations in how the game is played, even before taking into account different sports. In some variations, the same player can appear in an unlimited number of fantasy teams. This is common in large leagues which allow thousands or even millions of participants, where it would be infeasible to place constraints on how many participants can have a certain player in their team.

In smaller leagues however, it is common that each player can appear in only one fantasy team in the league. In such cases, it is necessary to begin the game with some method of working out which managers get which players for their fantasy teams. The simplest method is known as the **draft**: managers simply take it in turns to select a player from those available, until all squads are complete. However, for those looking for an extra layer of strategy, the **auction** is preferred. In an auction, each manager starts out with a fixed budget, and must assemble a squad of players by bidding against other managers. The auction is the method of squad selection which has been chosen for this application.

Another distinction to draw is between **daily** and **season-long** fantasy sports. In season-long games, the idea is to pick a squad that will score the most points over an entire season, typically over many months. Mid-season changes to squads are often permitted in these games. In daily games, the aim is to pick a squad

which will score well in one specific event. Despite the name, this event may last for a few days (for example, a four-day golf tournament, or a round of football fixtures spread over a weekend). In such games, once the initial squad is selected, it typically cannot be changed. Point-scoring is typically also more granular in daily games. In football for example, a season-long game might only award points for goals scored or assisted, but a daily game will additionally award a small number of points for more common occurrences such as completing a pass, or a successful tackle. This application is intended for daily games, although most of the logic could apply to either and could be adapted.

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. [2]

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.

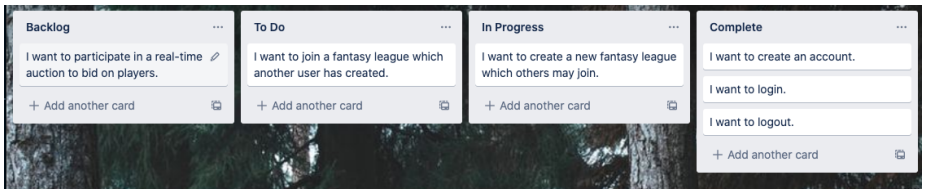

Figure 1: Trello Board

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

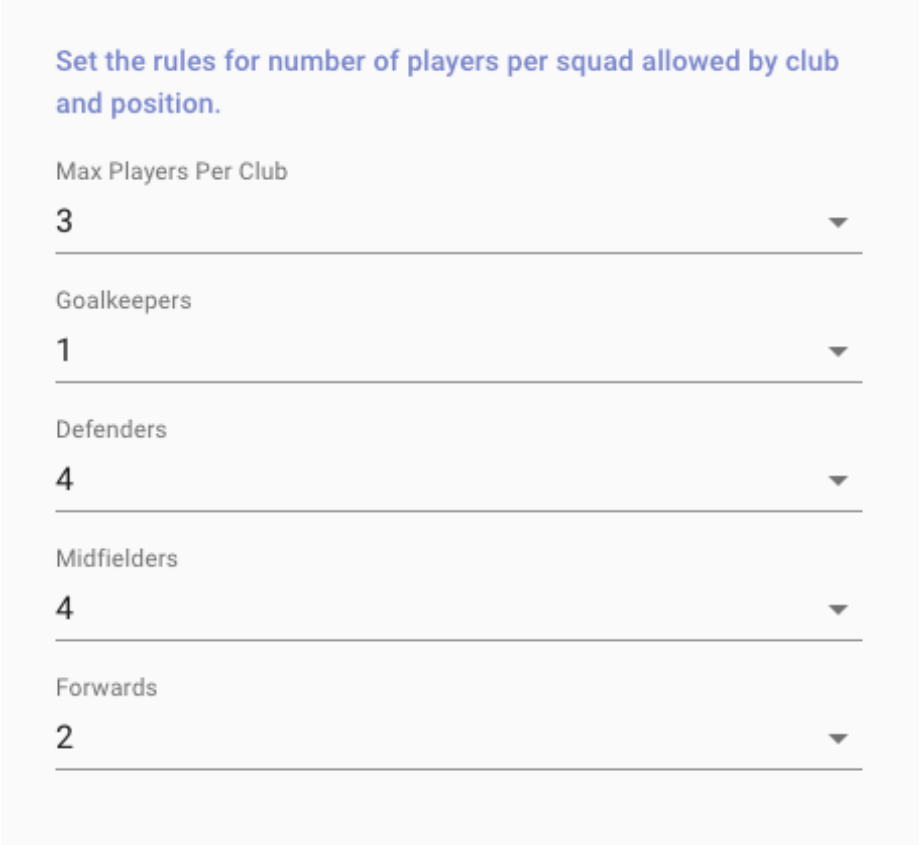

Figure 2: League Creation Screen

seen in figure 3.

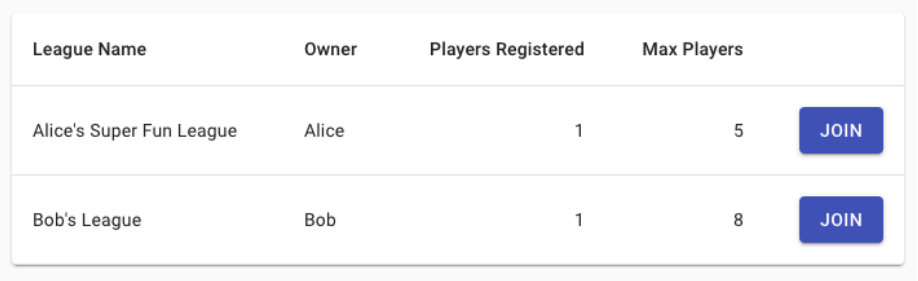

Figure 3: Available Leagues Screen

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 consumption on a mobile device, but works well on a large tablet (e.g. 10.2" iPad) or bigger. 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[3] 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 developer 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 for this application. 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[4]. 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.[5]

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.[6] 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

Visual Studio Code is a feature rich text editor as opposed to a fully-fledged integrated development environment. Some of its useful features can be seen in figure 4.

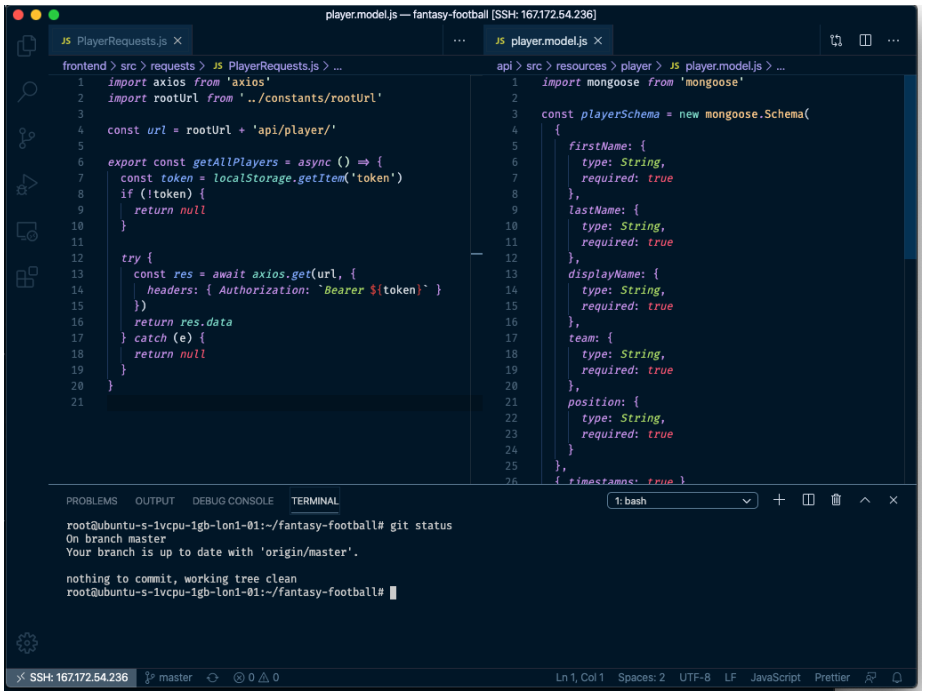

Figure 4: Visual Studio Code

In the bottom left, a status bar shows that the current working directory is actually a remote server accessed via SSH (Secure Shell). This made working on the development server as simple for the developer as working on the local machine.

The integrated terminal is another useful feature, both for local and remote development environments. In the screenshot a git command is shown, which is only one of many uses for the terminal during development.

Two source code files are shown in the editor, side-by-side. In the left panel, there is client-side code for requesting a list of all available players, and on the right panel there is server-side code showing relating to the player model - the ability to view files side-by-side in this manner is extremely useful during development. In this example, the developer working on the client-side code is able to view the server-side code to learn what data structure to expect in the response.

The code in the screenshot (and all code in the application) is consistently

formatted according to rules chosen by the developer. This is handled by an extension for Visual Studio Code called **Prettier**. This frees the developer from having to worry about spacing and indentation - it is handled automatically on each save.

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 dependencies are required, so that when the code is deployed on a new machine, the process of installing these dependencies is automated. NPM (Node Package Manager) is very similar and would have been a suitable alternative.

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 is first and foremost a REST API. It receives requests from clients, performs the necessary database operations, and returns a response. The real-time bi-directional communication added using Socket.IO is important for the user experience, but it is worth noting that even if this functionality was removed, the application would still work. The user experience would be terrible - they would have to constantly refresh their page during the auction to see if there had been any new bids, but with enough persistence from the users, the auction could be completed correctly. There are many good reasons for this approach, described in the following section.

REST API

A REST API codebase is typically well-organised and simple for the developer to navigate. There are other ways to structure a project, but in this case, the

approach which was taken can be seen in figure 5. Each entity (or resource) has its own directory, and within that directory is a file each for **model**, **controller** and **router**:

- The **model** file contains the schema information determining the structure of the objects to be stored in the database.
- The **controller** file contains functions for reading from and writing to the database.
- The **router** file defines how the various types of requests (e.g. GET and POST) are handled.

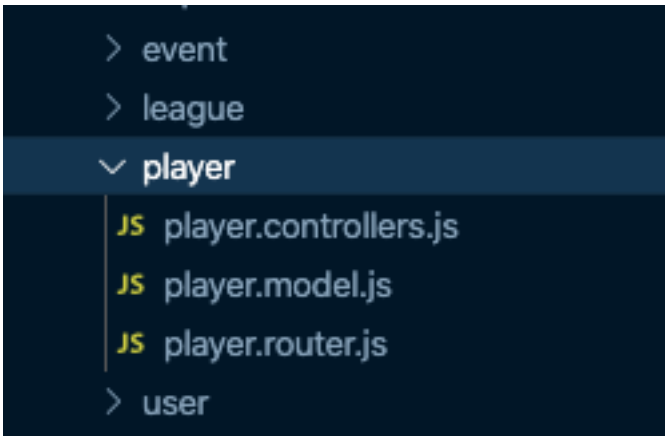

Figure 5: Resource Directory Structure

Another benefit of building the backend as a REST API is the ability to test each endpoint. Figure 6 shows an example of a simple request to return data for a specific league, which has successfully returned the desired JSON data.

This design also made sense when considering the necessary business logic. In the context of this application, the only time the server should be pushing data to clients which have not submitted a request, is after some change has been made to the database. This might be after a new bid, or bidding on an auction item has ended. However, data should never be emitted to all auction participants in the event of a failed bid, or before the bid has been successfully registered in the database. There is no client-to-client communication required or desired, as might be the case for a simpler use case like a chat room. With this in mind, implementing all of the business logic in the form of a REST API, and emitting updated data to clients as a side-effect using Socket.IO, seemed like a good solution.

Login System

The REST API routes needed to be protected so that only authenticated users could gain access. This is true of almost every REST API, so rather than

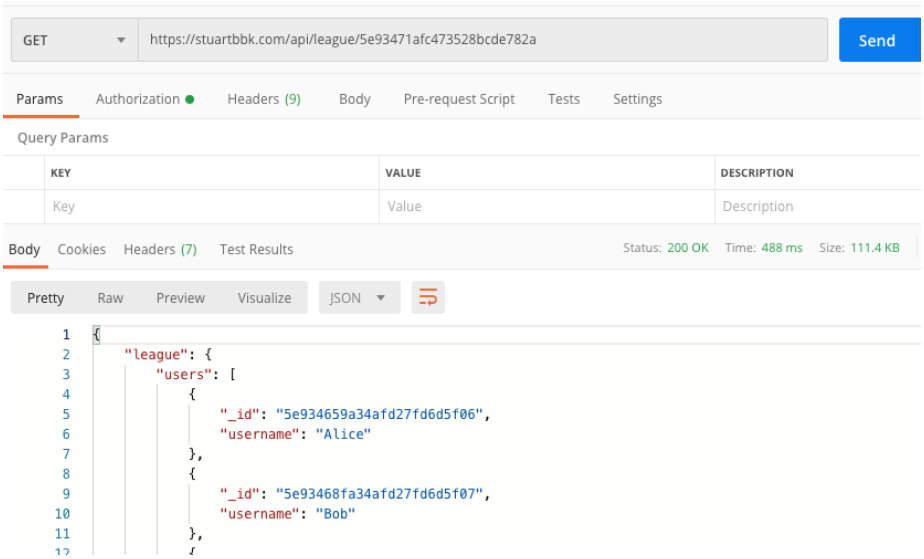

Figure 6: GET Request Testing

design a login system from scratch, an implementation was copied from the repository[7] for a REST API design course[8] available on the FrontendMasters website. Minor edits aside to tailor the solution to this application, the credit for the code in the auth.js and user.model.js files belongs to the teacher of said course, Scott Moss.

The login system solution generates a **JSON Web Token** on each login or account creation, and returns that token to the client. The client must then use this token to authenticate themselves when accessing any other resources.

Socket.IO Integration

The Socket.IO library offers features for organising and managing sockets in an application using **namespaces** and **rooms**[9].

Namespaces allow for separation of concerns between communication channels in an application. In this application, only one namespace ('leagues') was required, but if it was later decided to, for example, add a chatroom to the home page, this could exist in a separate namespace, keeping the application logic for different parts of the application separate.

Each namespace can contain several rooms. In this application, a separate room exists for each league, so that once a client has subscribed to the room, the server can push messages to the client. The business logic of this application was implemented in such a way that the server pushes messages to all clients in the room after some database action has been performed, to ensure that all

clients have the most up-to-date representation of the league state, without them having to request it manually.

Although Socket.IO also allows the client to emit messages to all other clients in the room, this functionality was not used in this application.

Data Model

Although a design for the data model had already been sketched out during the design phase, there were still some decisions to make relating specifically to the MongoDB implementation. In MongoDB parlance, there are **collections** and **documents**. A collection can be considered analogous to a table in a relational database, and a document is a record within that collection. Each document is assigned a unique object ID, which acts like a primary key in a relational database table. MongoDB does not require that all collections enforce a schema, although in this application a schema was enforced. An example of a collection from this application is **players**, and each document within that collection represents a single player, as seen in figure 7.

Figure 7: Two Documents in Players Collection

MongoDB offers two different methods for modelling relationships between documents, both of which were utilised in this application:

- **Document References**[10] - this method uses references to object IDs to describe the relationship. This is similar to the way that a foreign key references a primary key in a traditional relational database. The

main benefit to this approach is that it avoids duplication of data, but the downside is that data from multiple collections may be needed to satisfy a query.

- **Embedded Documents**[11] - this method instead sees documents stored within other documents. With this approach, duplication of data may occur, but the number of read operations required to retrieve a document is minimised.

It is not always immediately obvious which method is best. Only with a strong understanding of how the application is going to use the data can an informed decision be made. Indeed, during the early stages of development of this application, it was necessary in one case to undo a lot of work and start over, after it was decided that the wrong approach had been chosen initially. Developers with more experience working with traditional relational databases may be attracted to the document references approach, but this can make life difficult when working with MongoDB.

During the early stages of development, most relationships were modelled using the document references approach, but problems started when it was necessary to update multiple documents in a single transaction. For example, during an auction it might be necessary for an operation to be performed which updates the main auction document, the auction users documents, and the available auction item documents. In order to ensure that there is no unintended behaviour, these updates must be performed in a single transaction, meaning that either all updates are successful or none are, with no other operations interleaved. Although MongoDB does offer multi-document transactions, the documentation[12] states:

> In most cases, multi-document transaction incurs a greater performance cost over single document writes, and the availability of multi-document transactions should not be a replacement for effective schema design. For many scenarios, the denormalized data model (embedded documents and arrays) will continue to be optimal for your data and use cases.

With this advice in mind, the schema was redesigned to use more embedded documents. All of the smaller subcomponents of the auction were added as embedded documents rather than references, and this made updates significantly more straightforwarded.

Uses cases for the document references approach still remained however - for example, modelling the relationships between the players collection and individual auctions. The full player list used for this application contains 619 players, and cannot be altered by the application. Therefore, there were no concerns regarding atomic update operations, so document references could be used to avoid duplicating all 619 player documents for each auction. Instead, each auction item simply refers to an object ID for the player it refers to.

The code for creating the schemas and performing database operations was done using a Node.js library called **Mongoose**. Mongoose is an object data modelling

library, which allows the developer to focus on modelling their data without concerning themselves with the complexities of the MongoBD query language. The resulting code is more readable, and allows the developer to easily see the structure of the data they will be working with. The code snippet below shows the schema for the current live auction item:

```
import mongoose from 'mongoose'
import { bidSchema } from './bidSchema'

export const liveAuctionItemSchema = new mongoose.Schema(
  {
    player: {
      type: mongoose.SchemaTypes.ObjectId,
      ref: 'player',
      required: true
    },
    bidHistory: [
      {
        type: bidSchema
      }
    ],
    currentHighBid: {
      type: Number,
      required: true,
      default: 0
    },
    currentHighBidder: {
      type: mongoose.SchemaTypes.ObjectId,
      ref: 'user'
    }
  },
  { timestamps: true }
)
```

It makes use of both document references (for the player and the user representing the current high bidder), and embedded documents (an array of bid documents representing the bidding history).

Auction Overview

Implementing the server-side logic for the auction was the most challenging and interesting part of the development process. Lots of validation and checking of constraints was required, as well as managing the countdown timer and sale of each item correctly. There are several different stages to the auction:

1. League creator starts the auction.
2. A user selects a player to be the next auction item.

3. Real-time bidding begins.
4. A countdown timer begins after the first succesful bid, and is reset after every subsequent successful bid.
5. When the countdown timer reaches 0, the player sale is finalised.
6. Steps 2-5 are repeated until all squads are complete, at which point the auction is finalised.

Each of the above steps is discussed in greater detail in the following six sections.

Starting the Auction

Once enough users have joined a league, the league's status field will be set to 'ready'. Only once the league is in 'ready' state, will the league creator be permitted to start the auction. When the creator sends a successful request to the server to start the auction, several database updates are required:

- League status is moved from 'ready' to 'auction'.
- The embedded auction document is prepared with the appropriate users and saved to the league document, as seen in the snippet below.

```
const auctionUsers = league.users.map(u => {
  return { user: u, squad: [], budget: defaultValues.startBudget }
})
league.status = 'auction'
league.auction = {
  auctionUsers,
  soldAuctionItems: [],
  liveAuctionItem: null,
  nextUser: user
}
await league.save()
```

The above snippet demonstrates another of the advantages of using the Mongoose library introduced in the previous section. It is possible to manipulate a document using plain JavaScript, before calling the `save()` method on the document to persist the changes to the database. This is significantly easier to read than a complex update operation, although these are sometimes necessary as will be seen in a later section.

After a successful database update, a message is emitted to all league users via Socket.IO (using the **rooms** functionality explained earlier). This message ensures that all clients receive a real-time update informing them that the auction has begun.

Next, a player must be selected for participants to bid on. The league creator is always responsible for nominating the first player. It might seem that random selection would be preferable to avoid giving an unfair advantage to the league creator, but there is no inherent benefit to being first to nominate a player. All

auction participants start with the same budget and ultimately the nominated player will be sold to whichever participant is willing to bid the highest.

Nominating a Player

The participant whose turn it is to nominate a player begins this process by submitting a request to select a certain player. Upon receiving this request, the server must perform some checks, to ensure that:

- The request was received from the expected authenticated user (the user referred to by the next user field in the auction document).
- The user is permitted to bid on this player (they may not be permitted if they already have too many players in their squad in the same position, or from the same club).

Assuming the request is successful, the auction document is updated with a newly generated live auction item embedded document, with the current user set as the highest bidder (with a bid of £0).

At this point, a countdown timer is triggered on the server, beginning a countdown to the player being sold. The countdown timer is explained in more detail in a later section.

In addition, a message is emitted to all league users in real-time, and bidding begins.

Bidding

This is where things get more complicated. Given the real-time nature of the auction with bidding open to multiple users, the server has to be able to handle two or more users attempting to bid on the player at approximately the same time, without compromising the integrity of the data.

On receiving a bid, the server must only accept it if:

- The user is permitted to bid on this player (the same club and position constraints as before).
- The user has sufficient budget to make the requested bid.
- The user is not already the highest bidder (it should not be possible for a user to outbid themselves).
- The bid amount is strictly greater than the existing highest bid.

That last check is the one which can be expected to fail in instances of two users attempting to bid at approximately the same time. In such cases of a bid being unsuccessful, an error response is returned to the unsuccessful bidder, and no updates relating to the failed bid are emitted to other users via Socket.IO.

For successful bids, database changes are required to reflect the new bid. The live auction item document is updated with the new highest bidder, highest bid

amount, and a new bid is appended to the array of bids representing the bidding history.

At this point, the countdown timer is reset, a message is emitted to all league users to inform them of the succesful bid, and bidding continues.

The difficult part of the above process is ensuring that the checking of constraints and updating of the database occurs in one atomic transaction. In the 'Starting the Auction' section above, a benefit of using Mongoose was discussed: the ability to manipulate documents as if they were plain JavaScript objects, with updated documents persisted to the database using the `save()` method on the document. Unfortunately, such an approach does not guarantee atomicity. In the milliseconds between the original document being retrieved from the database, and the edits being performed in plain JavaScript before being saved, it is possible that a new bid could arrive. This new bid could have constraints checked against the out-of-date document in the database, and this may lead to a bid being incorrectly accepted.

Therefore, it was necessary to use the `findOneAndUpdate()` method available on all Mongoose models, to perform validations and updates in a single atomic transaction. The resulting code snippet, appended below, is far less readable than the previous snippet, but this was a necessary evil to ensure that the integrity of the data does not become compromised when faced with multiple competing bids.

```
const league = await League.findOneAndUpdate(
  {
    _id: leagueId,
    status: 'auction',
    users: { $eq: user },
    'auction.liveAuctionItem._id': auctionItemId,
    'auction.liveAuctionItem.currentHighBidder': { $ne: user },
    'auction.liveAuctionItem.currentHighBid': { $lt: amount }
  },
  {
    'auction.liveAuctionItem.currentHighBid': amount,
    'auction.liveAuctionItem.currentHighBidder': user,
    $push: { 'auction.liveAuctionItem.bidHistory': { user, amount } }
  },
  { new: true, useFindAndModify: false }
)
```

The first argument to the `findOneAndUpdate()` method above is an object containing query filters, and the second argument is an object containing update instructions (the third argument containing options is not of importance to this discussion). It can be seen that one of the items in the query filters compares the current high bid amount to the new bid, with a `$lt` (less than) operator. This means that if a higher (or equal) bid has already been registered, the query

will find no results, and no erroneous attempt to update the document will be made. An unsuccessful bid error message will be returned to the bidder.

It is worth noting that not all constraints are checked by this single operation. Some of the validations can be performed in advance (club, position and budget), so they are not shown in the snippet above. Only those validations which are extremely time sensitive are included in the atomic `findOneAndUpdate()` operation. A player's club, position and budget constraints cannot be affected by a new bid.

Countdown Timer

As noted in the preceding two sections, a bid on a player triggers a countdown timer, starting at 10 seconds. Any subsequent successful bid resets this timer back to 10 seconds. After 10 seconds of inactivity (no new bids), the sale is finalised.

Node.js features a convenient built-in function called `setInterval()`, which accepts two non-optional arguments: a function to be executed repeatedly, and a number representing the delay between executions of said function in milliseconds. A related function, `clearInterval()` is also provided, to allow the application to terminate the repeated execution initiated by `setInterval()`. Both of these functions were necessary to implement the countdown timer in this application. `setInterval()` is called each time a new bid is successfully registered, to start a new countdown timer at 10 seconds. `clearInterval()` is used to ensure that old countdown timers for bids which are no longer the highest bid are terminated.

A code snippet detailing the use of these functions in the application can be seen below. Some details are redacted for the sake of clarity; the complete version can be seen in the `startCountdown()` function in the file auction.js.

```
let count = defaultValues.countdownTimer
const countdown = setInterval(async () => {
  const league = await checkBidIsHighest(leagueId, auctionItemId, amount)

  if (!league) {
    return clearInterval(countdown)
  }

  count -= 1
  if (count <= 0) {
    clearInterval(countdown)
    // Redacted: some code to finalise sale of player
  }

  socketIO.to(leagueId).emit('countdown', count)

}, 1000)
```

The snippet above handles the following sequence of events, which occurs for each successful bid:

- The `count` variable is initialised to the default starting value (10 seconds in this case).
- A call to `setInterval()` initiates the repeated execution of a function to be called once every 1000 milliseconds.
- Inside the function, the database is queried to check that the bid amount is still the highest bid.
- If the check above fails, this means that a higher bid has subsequently been registered, and `clearInterval()` is called to terminate the countdown timer for this bid which is no longer the highest.
- If the check instead confirms that the bid is still the highest, execution of the function continues. The countdown timer is decremented by 1 second.
- If the countdown timer has not yet reached 0, an update is pushed to all league users via Socket.IO.
- If the countdown timer has reached 0, `clearInterval()` is called to ensure no further executions of this function, and some code to finalise the sale of the player is executed, the details of which are reserved for discussion in the following section.

Auction Item Sold

As mentioned in the previous section, once the countdown timer reaches 0, some code to finalise the sale of the player is executed. This process involves a significant amount of data manipulation:

- The winning bidder's squad and budget are updated to reflect the sale.
- The player sold is added to the array of sold items.
- The currently live auction item is set to `null`.
- The next user to nominate a player is selected (this is fairly complex as it involves checking to see which users still have incomplete squads).

A check is also completed to work out whether or not this player sale signifies the end of the auction entirely, but for the purposes of this section, it is assumed that the result of this check is negative. The completion of the auction is discussed in the next section instead.

In the 'Bidding' section above, it was established that care must be taken to ensure that no new bids can be registered while database updates are in progress. This remains true when finalising the player sale, but the data manipulation required for this step was prohibitively complex to consider implementing as a single atomic update operation, so an alternative approach was taken.

Before beginning the complex updates, a single atomic update operation to set the league status from 'auction' to 'locked' is executed. All bid handling code is implemented to only accept bids if the league status is set to 'auction', so this has the desired effect. For the length of time that the league status is set

to 'locked', complex updates can be performed using plain JavaScript, in the manner demonstrated in the 'Starting an Auction' section. These updates are not atomic, but it doesn't matter as long as no other database operations are permitted when the status is set to 'locked'. Once the updates are complete, the league status is set back to 'auction', so that the auction can continue.

Two separate messages are emitted to all league users via Socket.IO during the above process. The first message is sent as soon as the league is locked to confirm the sale. A second message is sent 3 seconds later to indicate that the next player should be selected for auction. This delay is deliberate to enforce a short pause between one auction item ending and a new one beginning. There is no technical reason which necessitates this delay (the database updates take only milliseconds), but rather it is implemented for reasons relating to the user experience, which will be discussed in more detail in the 'Frontend' section.

Auction Complete

After each player sale is finalised, a check is completed to determine whether or not all squads have been filled. When the result of this check is positive, some further data manipulation is completed to finalise the auction. Due to the use of the 'locked' status introduced in the previous section, this process is fairly straightforward from a technical standpoint, with no potential for conflicts. The following updates are made to the league document:

- The status field is set to 'postauction', to indicate that the auction is complete.
- Some new data structures are created to prepare for the next stage of the game, based on the results of the auction:
  - A new embedded document to track data relating to each user's points scored and their rank in the league.
  - Contained within this document for each user, an additional embedded document for each player in their squad, to track points scored by individual player.

At this stage, all scores are initialised to 0, but once the event (round of real-life football fixtures) is underway, these will be updated.

Post-Auction Points Scoring

In order for the game to display updated points based on real-life football data, this would require either that an administrator manually updates scores, or that the application has the ability to consume data from a third party football statistics API. At this time however, the application simply randomly generates points every few seconds, to simulate events. The result is that the points scoring portion of the game lasts for only about a minute rather than hours or days, as it would if played for real. This allows for the functionality of the game to be demonstrated in a short space of time without requiring significant manual data entry.

The sequence of events for this randomly generated points-scoring process is as follows. Most steps would remain the same for a version which used real data instead.

- An administrator triggers the simulation of an event by sending a POST request to the server.
- The event's status is updated to 'live'.
- Points for each player are randomly generated every 3 seconds, and the event document is updated accordingly.
- After each event update, a message is sent to each league which is using this event to track its points-scoring.
- For each league, individual player points are taken from the event data, and total score for each league user is calculated by summing points for each player in their squad. Live updates are pushed to each league user via Socket.IO messages.
- After the 20th points update, the event status is set to 'complete', and all leagues tracking this event are also set to 'complete'. One final update is pushed to each league user, confirming the completion of the game.

Any league which has not yet completed its auction at the time of the event starting is automatically cancelled.

Frontend

Testing

User Acceptance Testing

Edge Cases

Summary

Reflection

Future Improvements

Bibliography

1. Green, C. (2014). Wilfred 'Bill' Winkenbach invented Fantasy Football way back in 1962 with GOPPPL in Oakland. Available at: https://web.archive.org/web/20150929163914/http://www.newsnet5.com/sports/wink-wilfred-bill-winkenbach-invented-fantasy-football-way-back-in-1962-with-gopppl-in-oakland.

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

3. 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/.

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

5. 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/.

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

7. Moss, S. (2020). API design in Node.js with Express, v3. Available at: https://github.com/FrontendMasters/api-design-node-v3.

8. FrontendMasters, S.M. / (2020). API design in Node.js, v3. Available at: https://frontendmasters.com/courses/api-design-nodejs-v3/.

9. Socket.IO (2020). Socket.IO Rooms and Namespaces. Available at: https://socket.io/docs/rooms-and-namespaces/.

10. MongoDB (2020). Model One-to-Many Relationships with Document References. Available at: https://docs.mongodb.com/manual/tutorial/model-referenced-one-to-many-relationships-between-documents/.

11. MongoDB (2020). Model One-to-Many Relationships with Embedded Documents. Available at: https://docs.mongodb.com/manual/tutorial/model-embedded-one-to-many-relationships-between-documents/.

12. MongoDB (2020). Atomicity and Transactions. Available at: https://docs.mongodb.com/manual/core/write-operations-atomicity/.