Fail-Safe Cloud Tournament Engine with Error Detection and Error Recovery

Kyle Chapman (20703236)
Supervisor: Dr. Cornelia P. Inggs
Cosupervisor: Mr. Andrew J. Collett

9 November 2020

Contents

Chapter 1

Introduction

The Fail-Safe Cloud Tournament Engine (FSCTE) is an extension of the Cloud Tournament Engine (CTE) developed by Reece Murray in 2018, which is an extension of the Tournament Engine (TE) built on the Ingenious Framework[1] (IF). The goal for the CTE is to allow people to play a turn-based game, provided it is supported in the IF, against each other in a match, where many matches make up a tournament. Multiple tournaments can be run to determine how good each player is. Tournaments can separated into two categories: private and public.

Private tournaments are tournaments that only the administrators can add players to, and they are mainly used to test some players against other players. Public tournaments are tournaments that any user can add their own players to and administrators can also add players to. These tournaments are used to put players against each other to rank each player based on how many wins they get against other players.

It was found that there were some major limitations to the CTE, such as minimal error reporting, lack of error recovery and an unmaintainable code base to name a few. As a result, there were situations that arose in which a player failure resulted in a failure of the entire tournament. This prompted the need for a version that was fail-safe and included error reporting and error recovery, as far as possible. The FSCTE aims to address these limitations and improve the overall system stability by being fail-safe, which means that if any error were to occur, the response to the error would cause as little harm to the overall system stability as possible.

[1]`https://bitbucket.org/skroon/ingenious-framework/src/master/`

1.1 Scope

The system used to manage the FSCTE is made up of multiple Docker[2] containers, which run each section of the FSCTE in their own separate environment. If some error occurs in a section of the system, it will not affect the other sections, since they are in separate environments.

The FSCTE uses the following three main sections that are run in their own Docker containers:

- **Web User Interface**: Allows a user to interact directly with the FSCTE.
- **Database**: Stores all the information relevant to the FSCTE.
- **Database Watcher (Golang)**: Watches the database for any changes, such as tournaments being started, and sends messages to the front-end accordingly.
- **REST API**: Facilitates file upload to, and download from, the web user interface, as well as CAS authentication for logging in with Stellenbosch University credentials.

The layout of the entire FSCTE system is further discussed in chapter 3, *design and implementation*.

1.2 Document Outline

This report will discuss the overview of the system in chapter 2, the implementation and design of the system in chapter 3, then onto the testing in chapter 4 and discussion of the future of the project in chapter 5. Finally, the conclusion will be presented in chapter 6.

[2]`https://docs.docker.com/engine/docker-overview/`

Chapter 2

Overview

This chapter will give an overview of the requirements for the FSCTE system and how the requirements have been met.

The CTE used in 3rd year at Stellenbosch University is designed to manage many Othello [1] tournaments, involving many players, concurrently. A user is able to view public tournaments, players, referees, schedulers, etc. and view relevant statistics for ongoing and completed tournaments. The admin, usually the lecturer, is able to add tournaments, schedulers and rankers and the students are only allowed to upload their players and add them to public tournaments.

2.1 Separate enviroments

Each section of the FSCTE needs to be in their own separate environment, so that if some error occurs, it impacts only that specific section and not the entire FSCTE. Docker was chosen for this purpose, since it is relatively easy to spin up a section in its own environment and redeploy that section if some error occurs.

The docker container is created by writing a `Dockerfile` that specifies which base environment you want to use (e.g. `ubuntu`, `node`, `neo4j` etc.) and the command you want to run in the container. The container exits when the command finishes, so in order to keep the container running forever, infinite loops are useful. This means that the command will only finish when the user forcefully stops the container.

Docker-compose[1] is used to spin up all of the docker containers used by the FSCTE, using their respective `Dockerfiles`, and allows for more control of how the container should be set up. Local directories can be mounted into the docker containers so that any files that are generated inside the container can be accessed

[1] `https://docs.docker.com/compose/`

outside of the container, on the host machine. This is useful for accessing the log files after a match has completed, so that it can be seen if the match was a success or not.

2.2 Web User Interface

The previous web user interface implemented in the CTE was not very user-friendly, which resulted in a lot of confusion for the students who used the CTE. The entire web user interface in the FSCTE was written from scratch using new technologies, such as Vue[2] and TypeScript[3], so that it was more visually appealing and easier for the lecturers and students to use. Students and lecturers (or admins) can log into the web user interface using Stellenbosch University Single-Sign On or by specifying their username and password if they already exist in the database.

The students are able to upload their players to the web user interface and join public tournaments with their players. The students can have as many players as they would like, and each of the players are added to the tournament standings once they join a tournament. This allows the students to see how their players perform relative to other players in the same tournaments.

The lecturers, or admins, are able to upload their own rankers, schedulers, referees and players to the web user interface. More will be explained about the rankers, schedulers and referees in chapter 3.2.5. The admins can also add their own players to public or private tournaments, as well as they can add a student's player to a public or private tournament. The admins can start and stop tournaments, as well as create new tournaments.

After a match has completed, either ending in a failure or success, the log files of that match can be accessed if the student or admin desires. This means that either the student or admin can see what moves both the players made and the final result of the match.

2.3 Software Stack

The FSCTE uses the following software stack:

- **Vue** - Front-end framework that is used to interact with the FSCTE.

[2]`https://vuejs.org/`
[3]`https://www.typescriptlang.org/`

- **Typescript** - Superset of JavaScript that adds types, which is useful for catching bugs before they make it into production.
- **Python** - Used to create the REST API that facilitates file upload to, and download from, the web user interface, as well as providing authentication for logging in using Stellenbosch University Single-Sign On.
- **Neo4j** - Database that is used to store all the information necessary for the FSCTE to work. Highly relational database that uses the Cypher[4] query language to get or set attributes.
- **Golang** - Used to schedule matches in tournaments concurrently, with less overhead than using Java.
- **Docker** - Tool to allow different components of the FSCTE to operate in their own separate environments.
- **Bash** - Used for writing scripts that will setup the FSCTE and download any necessary programs.

The host system does not need to have docker pre-installed, since the FSCTE will check if docker and docker-compose are available in the setup script. If one or more programs are not installed, the script will download the necessary version onto the host machine.

[4]https://neo4j.com/docs/cypher-manual/current/

Chapter 3

Design and Implementation

This chapter will describe the design and implementation of all the components present in the FSCTE. Figure 3.1 shows an overview of the components present in the FSCTE and how they interact.

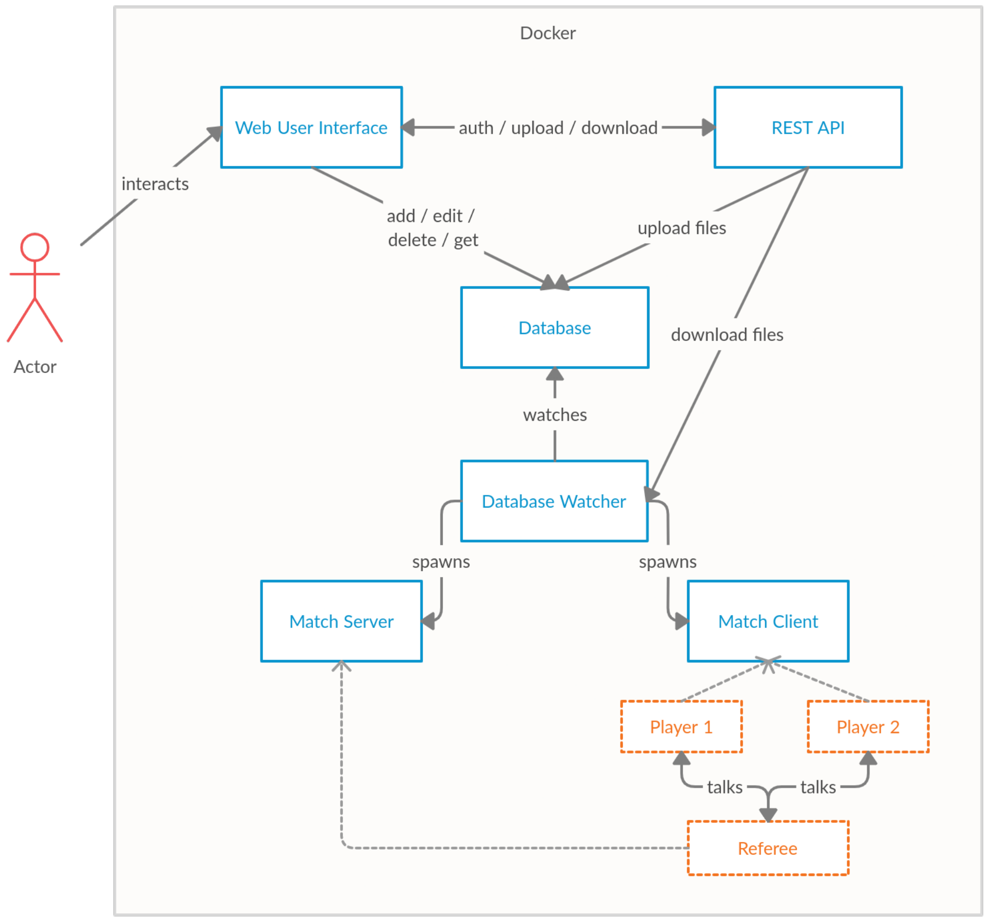


Figure 3.1: UML diagram representing the structure of the FSCTE

3.1 Overview

This section serves as an explanation for what can be seen in the UML diagram in Figure 3.1.

The actor represents a user, who can be either an admin or student. This user only interacts with the web user interface to add players, tournaments and anything the user desires, provided they have the correct permissions. There is a difference between a user and a player, where the user is the person who is currently logged in (student or admin) and the player is an executable that the user uploads to take part in tournaments. As such, a single user can have many players.

The rectangles with a blue outline represent docker containers and the rectangles with a dotted orange line represent entities inside the docker containers. The dotted grey line represents the docker container in which an entity resides.

3.2 Web User Interface

This section serves as an explanation for the purpose of the web user interface docker container seen in Figure 3.1.

The web user interface is designed to be a Single-Page Application (SPA) that uses components to render specific “pages”, instead of the traditional way of rendering complete HTML pages when the user navigates to a different URL route. This is accomplished by swapping out certain HTML elements with other elements so that the content of the page changes without having to re-render the elements that are present in both pages. As such, the performance of a SPA is significantly better than a traditional web app because of the reduced rendering needed to display different pages. Vue[1] was chosen as the framework to use because it is extremely easy to use and provides support for creating a SPA. An example of the login screen for the web user interface can be seen in Figure 7.1 in the Appendix.

The web user interface makes use of cookies to store the session, so that the user stays logged in even when closing the web user interface browser tab. The cookies are saved for seven days, meaning that once the cookies expire, the user has to log in again. Since Vue was chosen as the front-end framework, the library `VueCookies`[2] was chosen as the way to store cookies in the web user interface. This library was easy to use and provided many customizable options, as well as

[1]`https://vuejs.org/`

[2]`https://www.npmjs.com/package/vue-cookies`

it is regularly updated.

The web user interface consists of different tabs that a user can click to navigate to different views. The tabs that can be accessed vary depending on the level of permission of the logged in user, i.e. admins will be able to access more tabs than students will be able to. However, some tabs are accessible regardless of if a user is logged in or not. These tabs are the ‘Tournaments’ tab, ‘Matches’ tab, ‘Standings’ tab and finally the ‘About’ tab. If user logs in as a student, the user is able to access the ‘Players’ tab as well as all of the tabs accessible when not logged in. If the user logs in as an admin, the user is able to access the ‘Components’ tab as well as all of the tabs accessible to a student. The above tabs will be explained in further detail from section 3.2.1 to 3.2.6.

3.2.1 ‘Tournaments’ tab

The ‘Tournaments’ tab is accessible by a user who is not logged in, a student and an admin. A screenshot of the page after clicking the tab can be seen in Figure 7.3 in the Appendix, when logged in as a student, and Figure 7.4 in the Appendix, when logged in as an admin.

User Not Logged In

The user is able to view all of the public and private tournaments that are taking place. An overview of the tournaments is provided where the name of the tournament, the current status of the tournament (either “Not Started”, “Running” or “Finished”), the maximum time limit for each move (in milliseconds) and the number of players currently in the tournament are displayed in a table. The table uses a live-updating system, meaning that the user does not need to refresh the page to see any updates to the tournaments, instead the updates are shown immediately as they happen.

The user can click on any entry in the tournaments table to be redirected to the specific tournament information of the selected tournament. On this page, the user can see the type of the tournament (either public or private), the status of the tournament, the maximum time limit per move, the referee used, the scheduler used and the ranker used. The user can also see the player names of the current players in the tournament. The ranker, scheduler and referee will be explained in detail from section 3.5.1 to 3.5.3.

A table showing the current players participating the selected tournament is present at the bottom of the selected tournament information page. The user can see the player names, not the names of the users, of all the players participating in the selected tournament.

User Logged In As A Student

The user is able to see and do everything described above, as well as being able to add their own player to a specific public tournament. If the current user has not uploaded any players, they can select a specific tournament from the table and they can add a player from within the tournament information page. This will create a player, link the player to the user and put the player into the selected tournament. If the user already has uploaded a player, or many players, they will be presented with a dropdown list of their own players, and the user is then able to select which player they would like to add to the tournament.

If the user has a player participating in the selected tournament, the user can see the name of their player and remove their player from that tournament if they wish to do so.

User Logged In As An Admin

The user is able to see and do everything described above, as well as being able to create a tournament and add students' players to both private and public tournaments. When creating a new tournament, the user must specify the name of the tournament, the type (either public or private) and the maximum time per move (in milliseconds). A check is performed on the name of the tournament, which makes sure the name is unique. If the name is not unique, the new tournament is rejected and the user must enter a unique tournament name.

If the user selects a specific tournament from the table, the user can add a player of their own or add a player from a student to the tournament. Only an admin can add players to a private tournament, but students are able to add their players to public tournaments. The user can start and stop the tournament as they wish, as well as edit the information of a specific tournament. When editing the tournament information, the user can change the name of the tournament, the type of the tournament, the maximum time limit per move, the referee, the scheduler and the ranker. A screenshot of the popup displayed when editing a specific tournament can be seen in Figure 7.6. The user can also remove any student player from the tournament, as well as being able to delete the entire tournament if the user wishes.

3.2.2 ‘Matches’ tab

The ‘Matches’ tab is accessible by a user who is not logged in, a student and an admin. A screenshot of the page after clicking the tab can be seen in Figure 7.7 in the Appendix, when logged in as a student, and the page is the same for when logged in as an admin or when not logged in.

User Not Logged In

User Logged In As A Student

User Logged In As An Admin

3.2.3 ‘Players’ tab

The ‘Players’ tab is only accessible by a user who is a student or an admin. A screenshot of the page after clicking the tab can be seen in Figure 7.7 in the Appendix, when logged in as a student, and the page is the same for when logged in as an admin or when not logged in.

User Logged In As A Student

User Logged In As An Admin

3.2.4 ‘Standings’ tab

User Not Logged In

User Logged In As A Student

User Logged In As An Admin

3.2.5 ‘Components’ tab

3.2.6 ‘About’ tab

3.3 REST API

This section serves as an explanation for the purpose of the REST API docker container seen in Figure 3.1.

The REST API is written in Python 3.7 and Flask[3] because it is very easy to set up. It runs in its own docker container and is available on port 5000. The

[3]`https://flask.palletsprojects.com/en/1.1.x/`

API allows for file upload to and download from the web user interface, as well as providing access points for CAS authentication.

File uploading is used for when a user uploads their player using the web user interface. The file is selected from the user's operating system and uploaded to the system that is hosting the web user interface. The path to the player is stored in the database so that the player files can be retrieved when needed. File downloading is used for when a user wants to download the log files after a match has finished. The path to the match log is generated by taking the tournament name and player name of the requested file, then finding the file on the system hosting the web user interface and finally downloading it to the user's system.

The CAS authentication is used to allow students and lecturers to log into the web user interface using their Stellenbosch University credentials. When the user clicks the 'SU Login' button on the Login page of the web user interface, they are redirected to the Single-Sign On page that Stellenbosch uses for authentication. If the username and password provided is correct, a ticket gets returned to the REST API, which then gets validated. If the ticket is valid, then the username is returned to the REST API. If the ticket is not valid, then no username is returned. After the validation process, the ticket is invalidated so that it cannot be used again. Once the REST API has the username of the user, it passes the username back to the web user interface, which checks if the username is that of a student or admin. It performs this check by looking at the username, where the username of a student is made up of all numbers and the username of a lecturer is made up of characters. This user is then added to the database so that the players associated with the user can be stored in the database.

The following endpoints are available for access through the REST API:

`/check`

- **Usage**: Ensures the REST API is accessible.
- **Methods**: `GET`
- **Returns**:

```
{ "message": "Connected successfully" }
```

`/login`

- **Usage**: Attempts to log the user in using Stellenbosch University credentials.

- **Methods**: `GET`
- **Returns**:

  ```
  {
      "username": <username>,
      "ticket": <ticket>
  }
  ```

`/clear`

- **Usage**: Logs the Stellenbosch University user out. The logout route could not be `/logout`, since the CAS authentication logout route was `/logout` and various variables needed to be cleared before logging the user out.
- **Methods**: `GET`
- **Returns**: `None`

`/upload/<fileType>/<fileName>`

- **Usage**: Uploads a `fileType` called `fileName` to the web user interface local system. The `fileType` can be either a referee, ranker, scheduler or player and the `fileName` is the name of the file to upload.
- **Methods**: `POST`
- **Returns**:

  ```
  { "message" : "File successfully uploaded" }
  ```

`/download/<tournamentName>/<fileName>`

- **Usage**: Downloads the log file called `fileName` from the tournament `tournamentName`.
- **Methods**: `GET`
- **Returns**: `File` object.

`/remove/<fileType>/<fileName>`

- **Usage**: Removes a `fileType` called `fileName` from the web user interface's local system, if it exists. Mainly used when a user deletes a player.
- **Methods**: `GET`
- **Returns**:

  ```
  { "message" : "Successfully deleted file" }
  ```

3.4 Database

This section serves as an explanation for the purpose of the database docker container seen in Figure 3.1.

Neo4j[4] was chosen as the backend database because the main advantage is the fact that it is a graph database. Graph databases, such as Neo4j, differ from relational databases, such as SQL, in how they connect data. Neo4j uses nodes and relationships to connect data, whereas SQL uses primary and foreign keys. The data that the system uses benefits greatly from using relationships between nodes to describe the data, since, for example, a user can have many players (relationship is user to player) that participate in many tournaments (relationship is user to player to tournament). Such data would require multiple foreign keys and get very complicated very quickly.

Neo4j uses the Cypher query language in order to write queries for the data in the database. The Cypher query language closely resembles English, making it easy to write complicated queries and return exactly the data that is needed. Database drivers were written in both TypeScript, for the web user interface, and Golang, for the database watcher. The database drivers allowed the web user interface and the database watcher to add, edit, delete and retrieve data from the database. An example of the graph database can be seen in Figure 7.10 in the Appendix.

3.5 Database Watcher

This section serves as an explanation for the purpose of the database watcher docker container seen in Figure 3.1.

[4]`https://neo4j.com/`

The database watcher was written in Golang because it is a language that handles concurrency very well, since the first version was released recently, March 2012 [2], and has concurrency at its core. Golang is being updated frequently, with the newest version being released in August 2020. Golang also has multiple advantages over Java, such as:

- Low overhead, since Golang does not use the Java Virtual Machine.
- Concurrency is built into the core language and simple to use compared to Java threads.
- Faster performance when compared to Java.

The above advantages are the reason that Golang was chosen over Java for writing the database watcher.

The main role of the database watcher is to watch for any database changes at a set interval. The main change that the database watcher is looking for is when a tournament is started or stopped. When this change occurs, the database watcher retrieves the name of the tournament and retrieves the ranker, scheduler, referee and players that are participating in that tournament. Information about the ranker, scheduler and referee is given from section 3.5.1 to 3.5.3.

3.5.1 Ranker

The ranker is used to distinguish between the good and bad players by comparing the rating of the two players, where every player starts on the same rating. For every win that a player has, their rating increases by some amount and for every loss that a player has, their rating decreases by some amount. The amount gained or lost decreases for every match that the player plays, meaning that eventually the rating will be a true reflection of the skill level of the player.

The ranker needs to be written in Golang so that it can integrate into the database watcher and the specification for writing the ranker can be found at `https://git.cs.sun.ac.za/20703236/fscte-docs/-/blob/master/guidelines/Ranker.md`. An example of a ranker that uses the Elo [3] rating system can be found in the `mock-data` directory in the root of the FSCTE repository and it is called `EloRanker.go`.

3.5.2 Scheduler

The scheduler is used to schedule matches between two players that play against each other in order to see who is the better player. After every player has played against every other player in the tournament, the overall win-loss ratio for that tournament can be calculated, which can then be used to distinguish between good, average and bad players. The win-loss ratio and the rating system can be used in conjunction to organize fair matches between players are of a similar skill level.

The scheduler needs to be written in Golang so that it can integrate into the database watcher and the specification for writing the scheduler can be found at `https://git.cs.sun.ac.za/20703236/fscte-docs/-/blob/master/guidelines/Scheduler.md` An example of a scheduler that uses a round-robin strategy to organize matches between players can be found in the `mock-data` directory in the root of the FSCTE repository and it is called `RoundRobin.go`.

3.5.3 Referee

The referee is used to monitor the moves that the two players make in a match and check if any player makes an invalid move or times out. One player sends a move to the referee, which then checks if the move is valid and only sends the move to the other player if the move made is valid. If the move made is invalid, or the player times out, the match will be forfeit and the last player to send a valid move becomes the winner.

The referee needs to be a `JAR` file that is compatible with a game type supported by the Ingenious Framework[5]. An example of a referee for the Othello game is provided in the `mock-data` directory in the root of the FSCTE repository and it is called `OthelloReferee.jar`.

3.5.4 Match Server

The match server docker container is started when a new tournament is started, which starts the Java server using the referee that the tournament uses. The match server lasts for only a single match and the log file from the perspective of the referee is stored on the FSCTE local system. A path to the match log file is stored in the database so that the referee log file can be downloaded from the web user interface at a later stage.

[5]`https://bitbucket.org/skroon/ingenious-framework/src/master/IngeniousFrame/src/main/java/za/ac/sun/cs/ingenious/games/`

3.5.5 Match Client

The match client docker container is started after the match server docker container is started, and is used to create the lobby for two of the players in the tournament as well as play out the match between the players. The way the two players are selected is defined by the scheduler that is used for the tournament and the rating update after the match has concluded is defined by the ranker used in the tournament.

The match lobby is created in the match client docker container and then the players are copied from the FSCTE local system into the match client docker container. The players are then connected to the match server and play against each other by sending their moves to the referee, which verifies if the moves are valid and if they are valid, then the referee sends the move to the other player. This process continues until either a player sends an invalid move or the game is played out until completion. The log files of each player are stored and the path to the log files are added to the database so that each player log file can be downloaded at a later stage using the web user interface.

3.6 Repository

The code is hosted on GitLab as a private repository and can be found at `https://git.cs.sun.ac.za/20703236/fscte`. The instructions on how to start all of the docker containers is in the README, which is accessible at `https://git.cs.sun.ac.za/20703236/fscte/-/blob/master/README.md`. All of the files present in the repository take up 36 Megabytes of space and consist of 11652 lines of code. Vue makes up 53.2% of the code base, with Go making up 22%, TypeScript making up 19.4%, Shell making up 4.1%, Python making up 0.9% and the rest of the languages making up 0.4% of the code base.

Chapter 4

Testing

Chapter 5

Future Work

This chapter covers the future work that could be implemented in the Fail-Safe Cloud Tournament Engine (FSCTE) to extend the current functionality. The conclusion will be provided in section 6.

5.1 Kubernetes

Kubernetes[1] can be used to manage the docker containers, so that if a container crashes, another instance of that container can be redeployed without hesitation. Due to time constraints, it was not feasible to add Kubernetes to the current FSCTE.

5.2 Other Game Types

In the current FSCTE implementation, only the `Othello` game is supported. Supporting other game types will allow the FSCTE to be more abstract and versatile. Due to time constraints, it was not feasible to attempt to support other types of games.

[1] `https://kubernetes.io/`

Chapter 6

Conclusion

The main requirements of the FSCTE that needed to be satisfied were:

- have effective error handling
- have effective debugging facilities
- scale with tournaments and players
- have a logical, clean and maintainable code base
- have a more user-friendly web user interface

All of the above requirements were satisfied. The error handling requirement was satisfied by making sure that at any point where an error could occur, it was decided if the error was serious or not. If the error was serious, then an error message would be displayed and the component that caused the error would be stopped. If the error was not serious, then an error message would be displayed, but nothing would happen to the component that caused the error.

The effective debugging facilities requirement was satisfied by logging all information that happens inside each docker container, so that if some error occurs, the user can look in the docker logs to see at which stage the error occurred. This allows the user to know exactly what was going on before the error occurred so that the user can pinpoint the issue and fix the problem.

The tournaments and player scaling requirement was satisfied by writing the code base in such a way that it performs very well for any number of tournaments and players. Golang was mainly used to allow scaling to be possible through the use of concurrent threads running different tasks at the same time.

The maintainable code base requirement was satisfied by providing extensive documentation throughout the code base, so that anyone who reads the code base can understand what is going on without any issue.

The user-friendly web user interface requirement was satisfied by ensuring that the user can navigate to any page or get any information they want in a maximum of two mouse clicks. A lot of time was spent making sure that the web user interface looked clean and visually appealing, as well as easy to use.

Working on the FSCTE for the past year has been a great challenge, but also a great reward. Initially, it was tough to understand what was going on in the CTE because of the lack of documentation and confusing way that some sections were implemented. After understanding the previous work, the toughest challenge was getting the players to play a match and communicate their results to the web user interface. However, it was a fun and informative experience.

Chapter 7

Appendix

7.1 Web User Interface

The images from Figure 7.1 to 7.9 are screenshots from the web user interface that the admins and students will interact with.

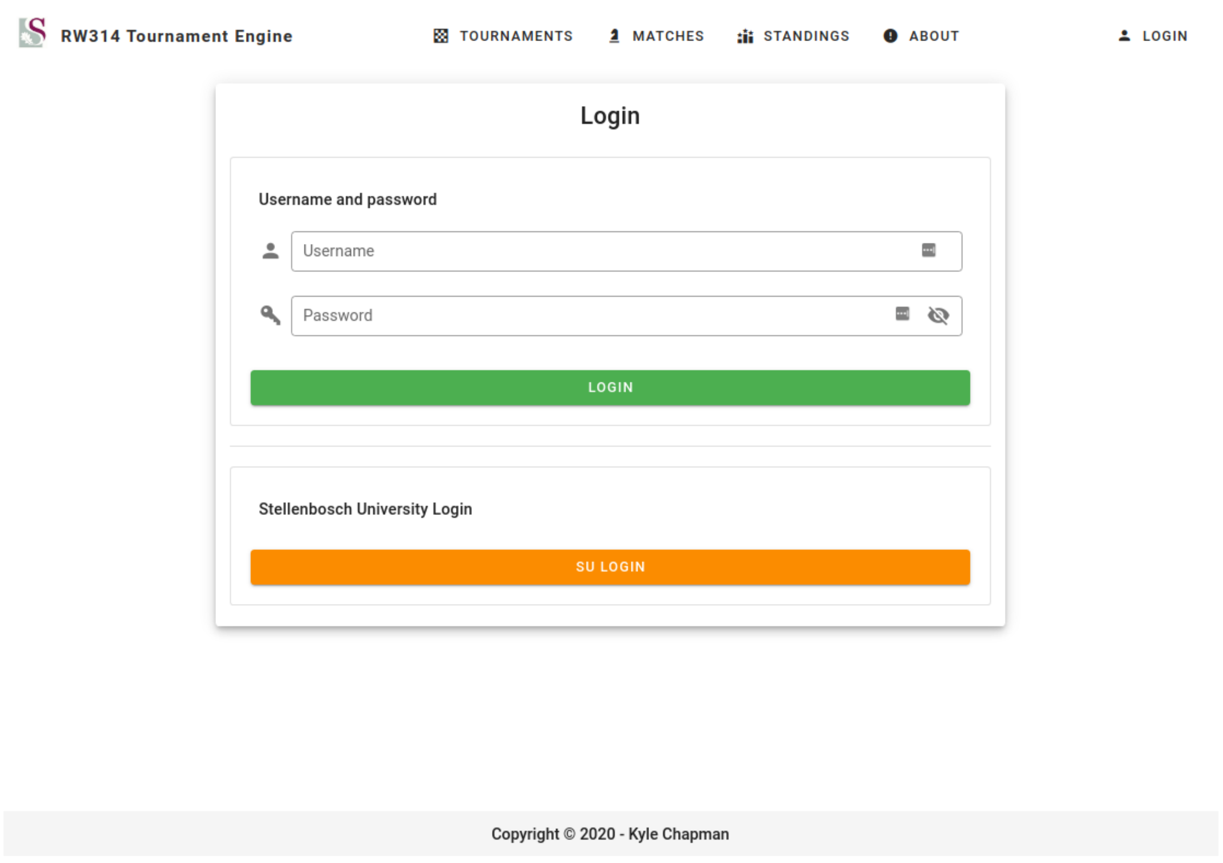
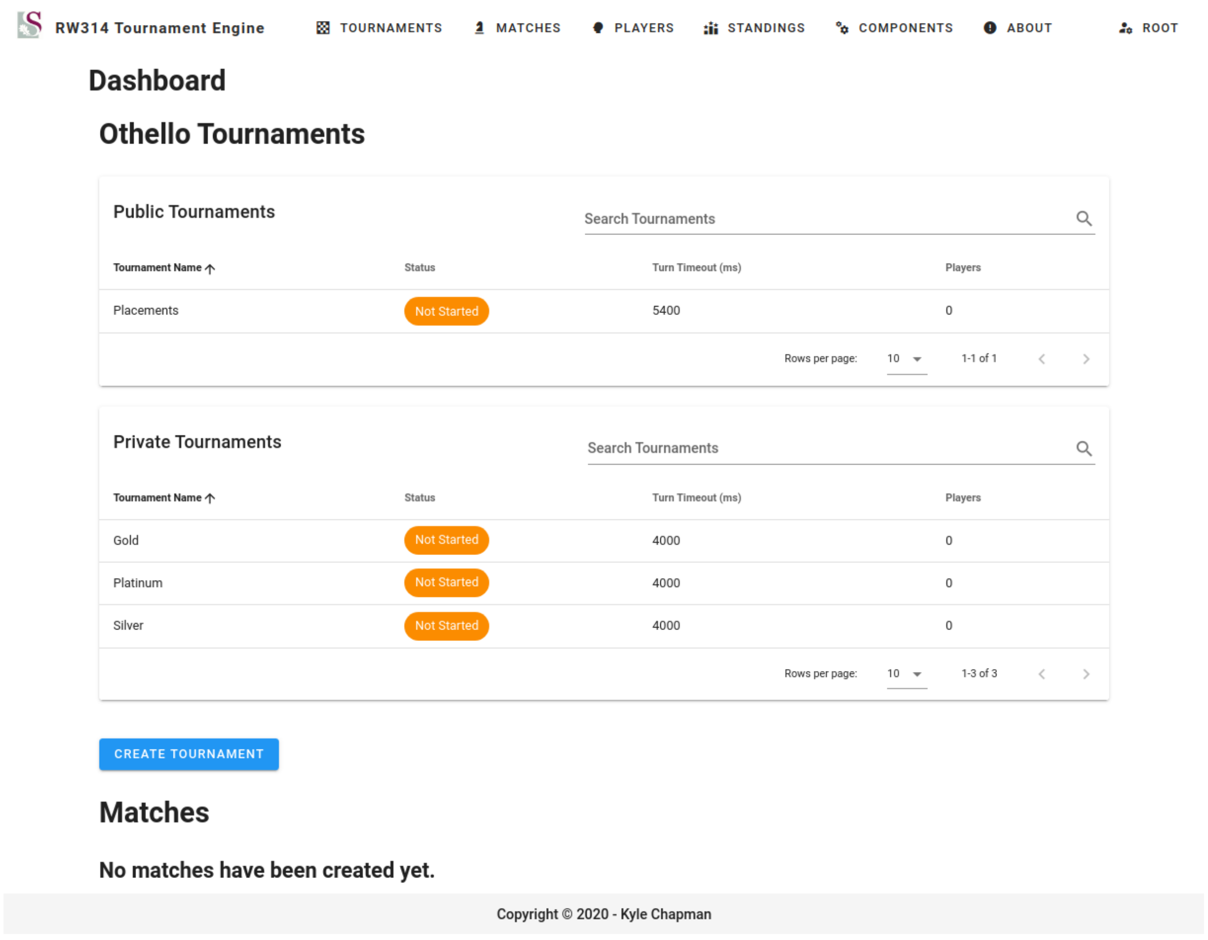


Figure 7.1: Login page



Dashboard

Othello Tournaments

Public Tournaments

Tournament Name ↑	Status	Turn Timeout (ms)	Players
Placements	Not Started	5400	0

Rows per page: 10 1-1 of 1

Private Tournaments

Tournament Name ↑	Status	Turn Timeout (ms)	Players
Gold	Not Started	4000	0
Platinum	Not Started	4000	0
Silver	Not Started	4000	0

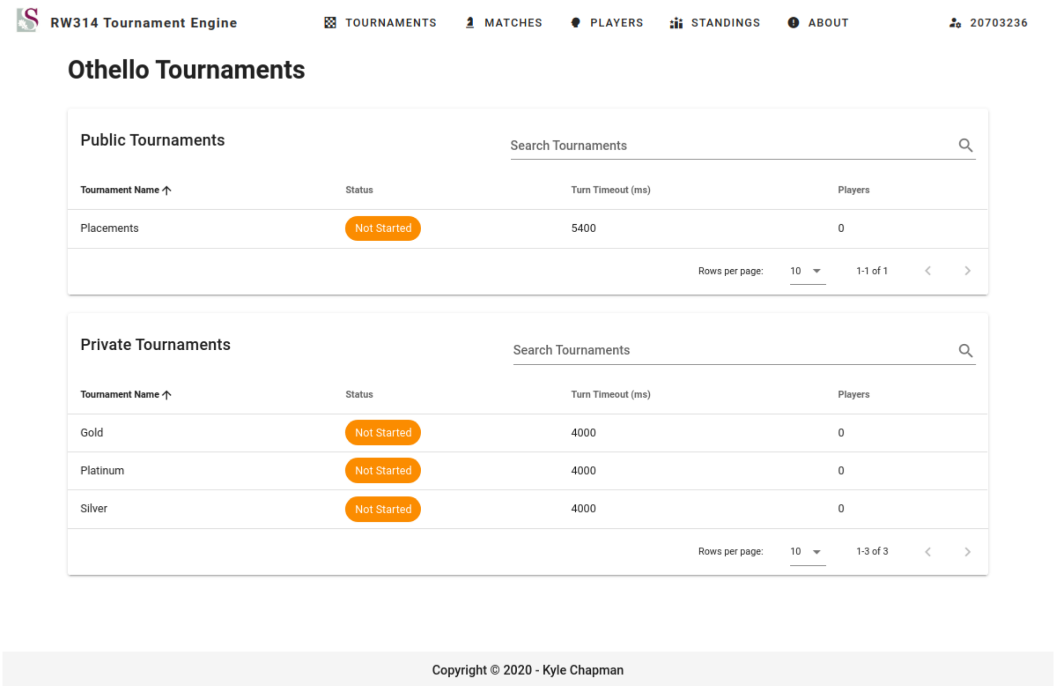
Rows per page: 10 1-3 of 3

Matches

No matches have been created yet.

Copyright © 2020 - Kyle Chapman

Figure 7.2: Dashboard after logging in as admin



Othello Tournaments

Public Tournaments

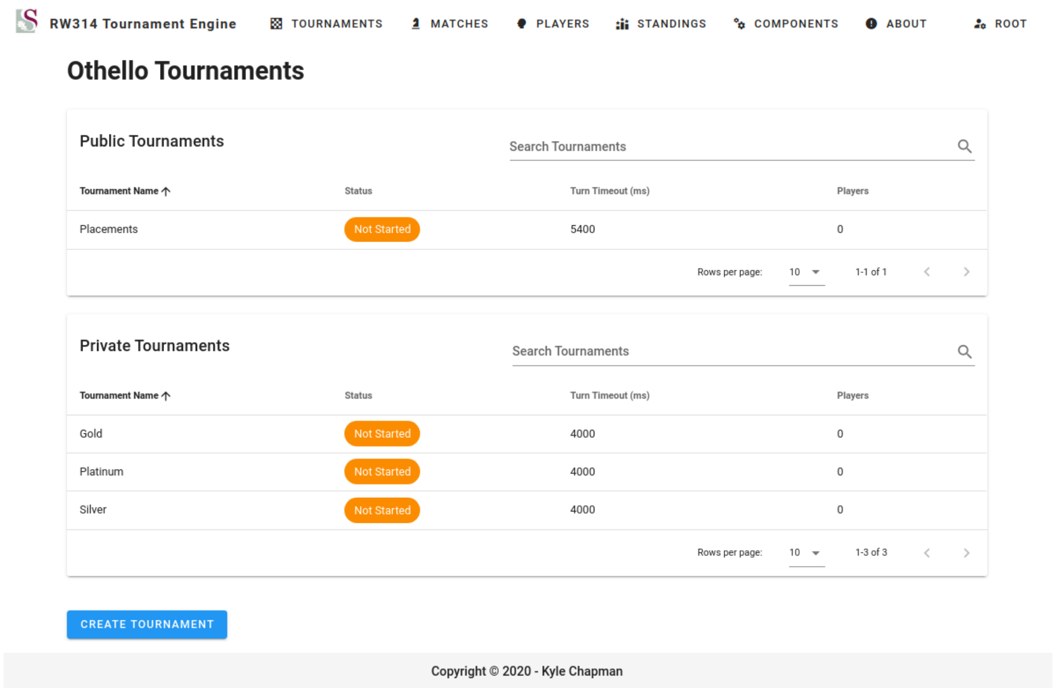
Tournament Name ↑	Status	Turn Timeout (ms)	Players
Placements	Not Started	5400	0

Private Tournaments

Tournament Name ↑	Status	Turn Timeout (ms)	Players
Gold	Not Started	4000	0
Platinum	Not Started	4000	0
Silver	Not Started	4000	0

Copyright © 2020 - Kyle Chapman

Figure 7.3: ‘Tournaments’ tab when logged in as a student



Othello Tournaments

Public Tournaments

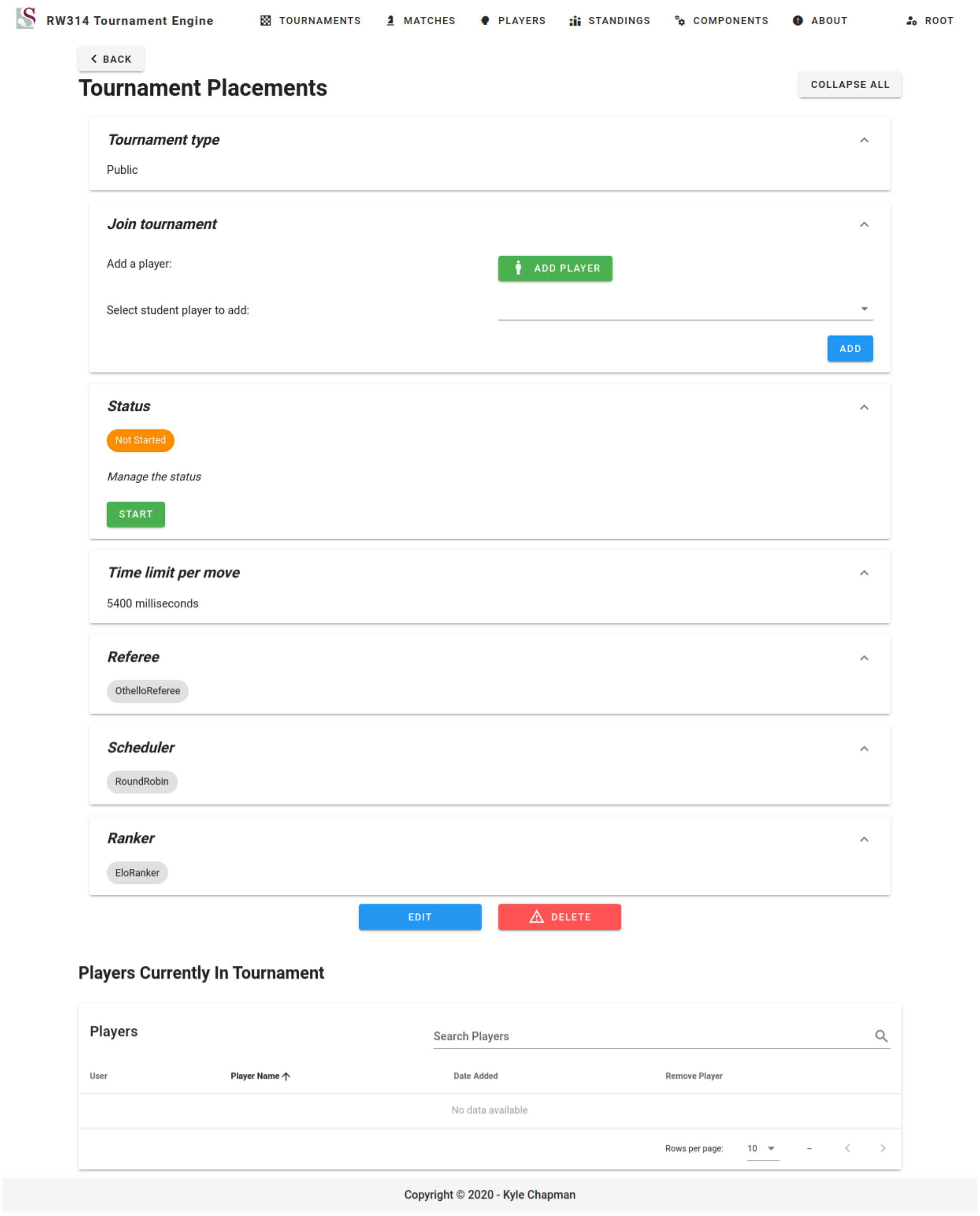
Tournament Name ↑	Status	Turn Timeout (ms)	Players
Placements	Not Started	5400	0

Private Tournaments

Tournament Name ↑	Status	Turn Timeout (ms)	Players
Gold	Not Started	4000	0
Platinum	Not Started	4000	0
Silver	Not Started	4000	0

Copyright © 2020 - Kyle Chapman

Figure 7.4: ‘Tournaments’ tab when logged in as an admin



Tournament Placements

Tournament type

Public

Join tournament

Add a player:

Select student player to add:

Status

Not Started

Manage the status

Time limit per move

5400 milliseconds

Referee

OthelloReferee

Scheduler

RoundRobin

Ranker

EloRanker

Players Currently In Tournament

Players

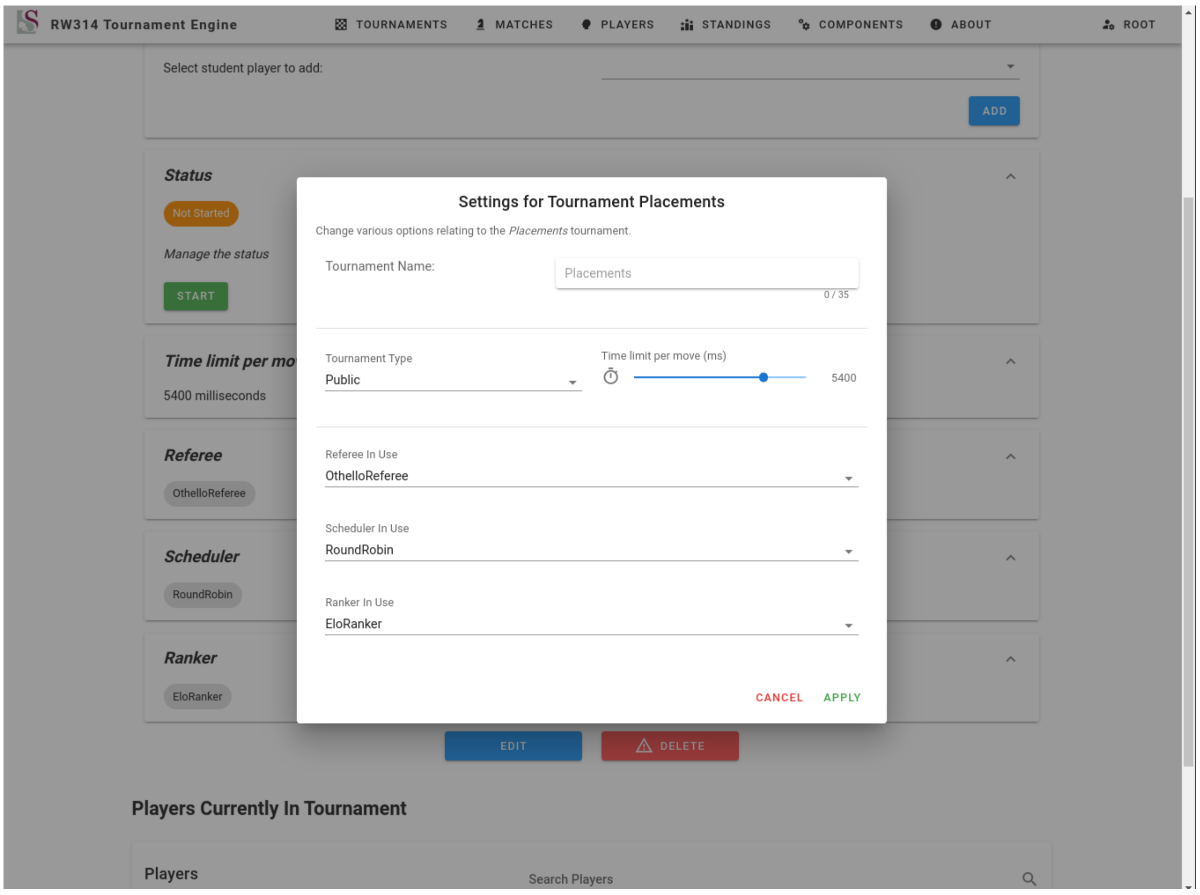
Search Players

User	Player Name ↑	Date Added	Remove Player
	No data available		

Rows per page: 10 –

Copyright © 2020 - Kyle Chapman

Figure 7.5: Page describing the tournament ‘Placements’



Select student player to add:

Status

Not Started

Manage the status

Time limit per mo

5400 milliseconds

Referee

OthelloReferee

Scheduler

RoundRobin

Ranker

EloRanker

Settings for Tournament Placements

Change various options relating to the *Placements* tournament.

Tournament Name: Placements 0 / 35

Tournament Type: Public

Time limit per move (ms): 5400

Referee In Use: OthelloReferee

Scheduler In Use: RoundRobin

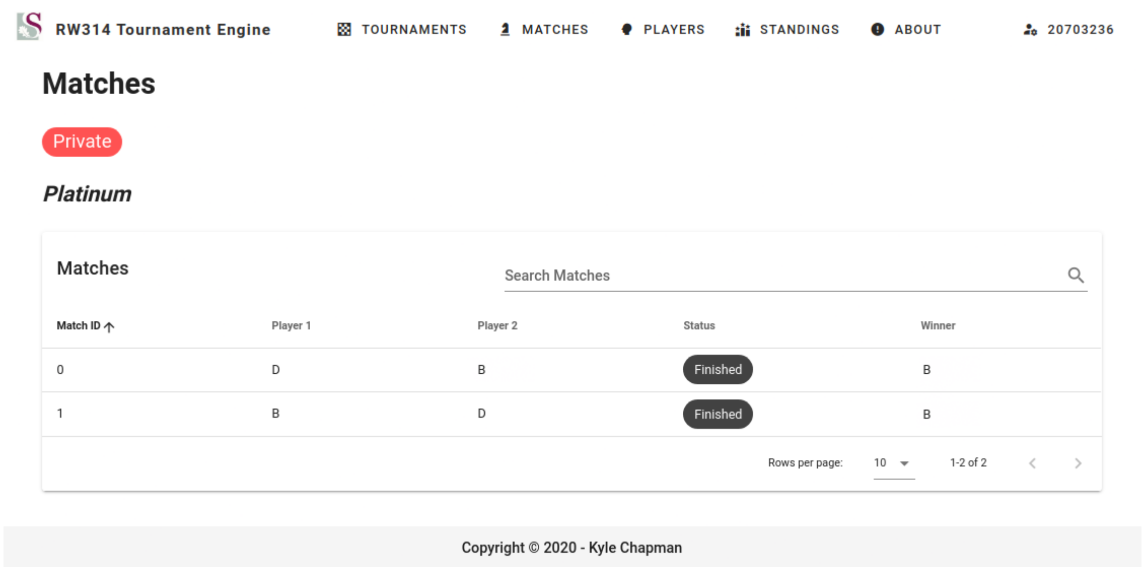
Ranker In Use: EloRanker

Players Currently In Tournament

Players

Search Players

Figure 7.6: Settings popup for the tournament ‘Placements’ when the logged in user is an admin



Matches

Private

Platinum

Matches

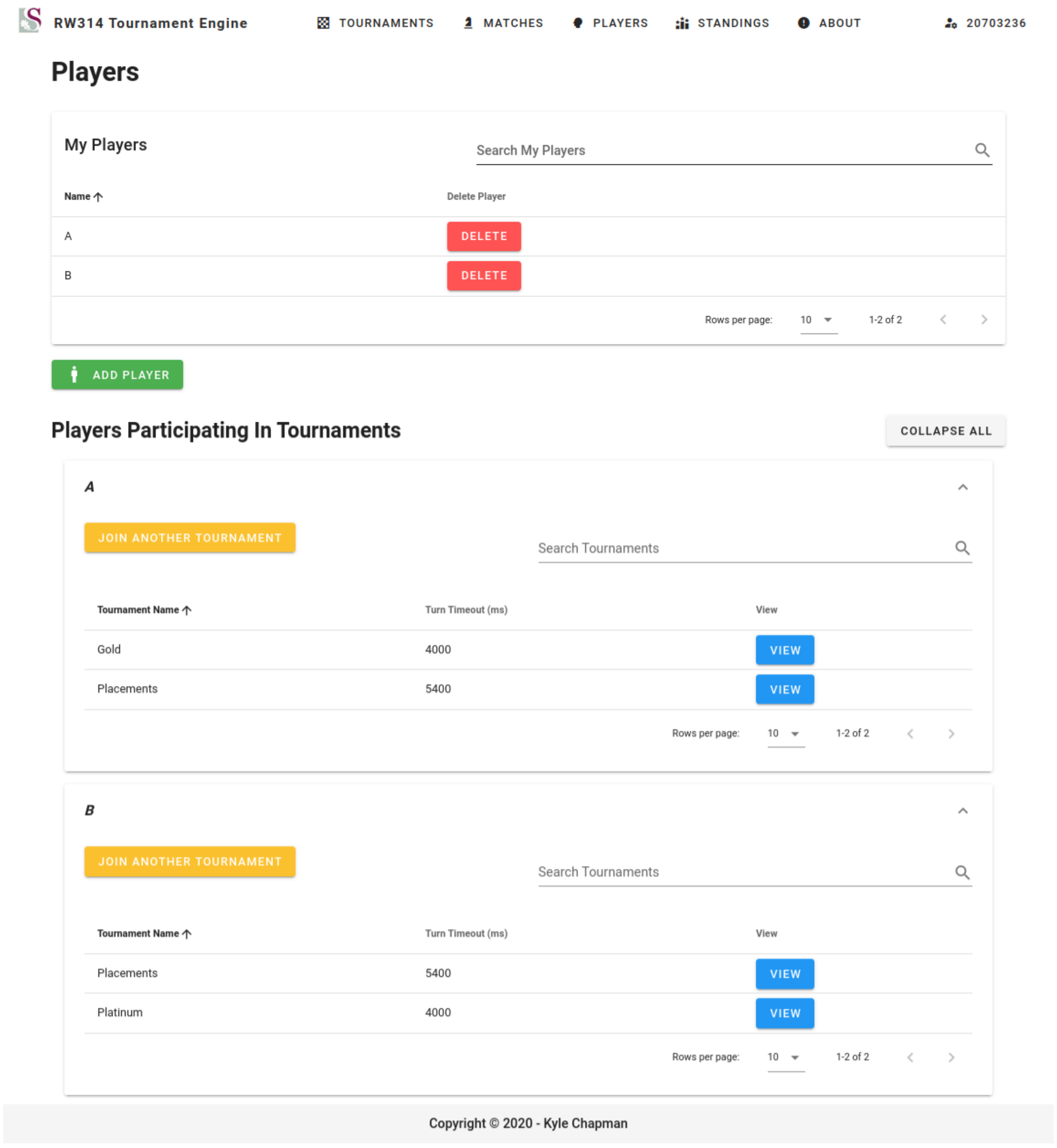
Search Matches

Match ID	Player 1	Player 2	Status	Winner
0	D	B	Finished	B
1	B	D	Finished	B

Rows per page: 10 1-2 of 2

Copyright © 2020 - Kyle Chapman

Figure 7.7: ‘Matches’ tab when logged in as a student



Players

My Players

Search My Players

Name	Delete Player
A	DELETE
B	DELETE

Rows per page: 10 1-2 of 2

Players Participating In Tournaments

A

Search Tournaments

Tournament Name	Turn Timeout (ms)	View
Gold	4000	VIEW
Placements	5400	VIEW

Rows per page: 10 1-2 of 2

B

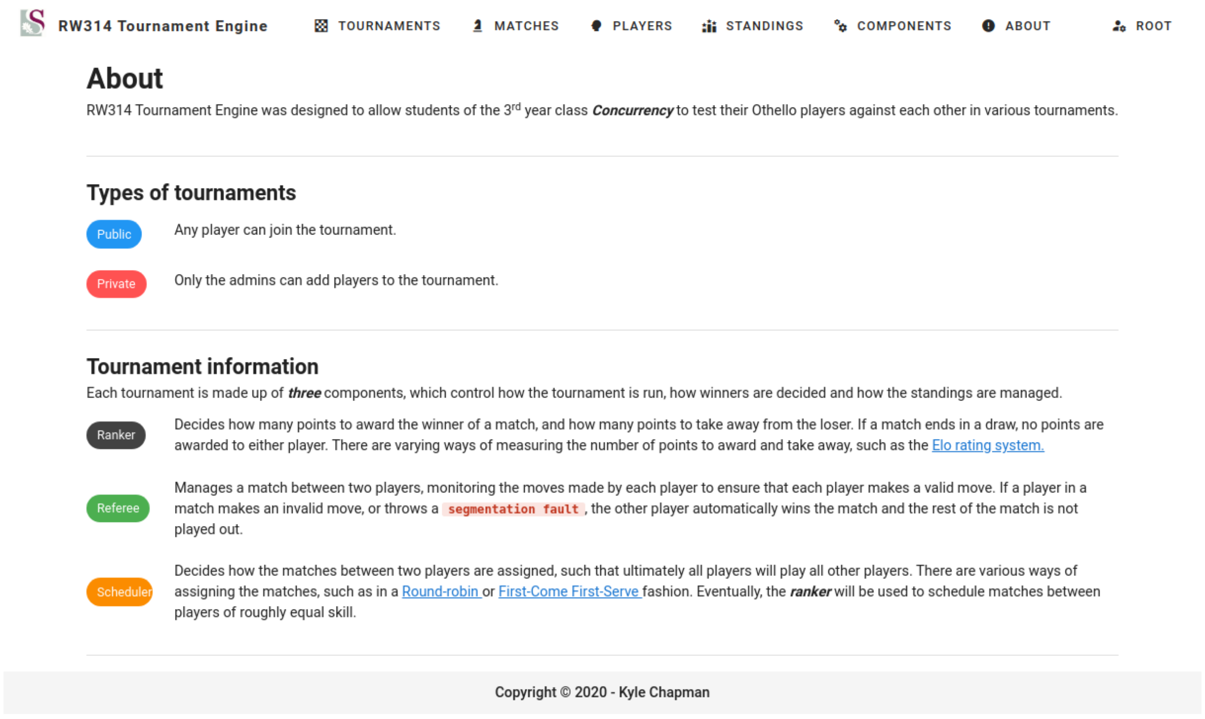
Search Tournaments

Tournament Name	Turn Timeout (ms)	View
Placements	5400	VIEW
Platinum	4000	VIEW

Rows per page: 10 1-2 of 2

Copyright © 2020 - Kyle Chapman

Figure 7.8: ‘Players’ tab when logged in as a student



About

RW314 Tournament Engine was designed to allow students of the 3rd year class ***Concurrency*** to test their Othello players against each other in various tournaments.

Types of tournaments

Public — Any player can join the tournament.

Private — Only the admins can add players to the tournament.

Tournament information

Each tournament is made up of ***three*** components, which control how the tournament is run, how winners are decided and how the standings are managed.

Ranker — Decides how many points to award the winner of a match, and how many points to take away from the loser. If a match ends in a draw, no points are awarded to either player. There are varying ways of measuring the number of points to award and take away, such as the Elo rating system.

Referee — Manages a match between two players, monitoring the moves made by each player to ensure that each player makes a valid move. If a player in a match makes an invalid move, or throws a `segmentation fault`, the other player automatically wins the match and the rest of the match is not played out.

Scheduler — Decides how the matches between two players are assigned, such that ultimately all players will play all other players. There are various ways of assigning the matches, such as in a Round-robin or First-Come First-Serve fashion. Eventually, the ***ranker*** will be used to schedule matches between players of roughly equal skill.

Copyright © 2020 - Kyle Chapman

Figure 7.9: About page

7.2 Database

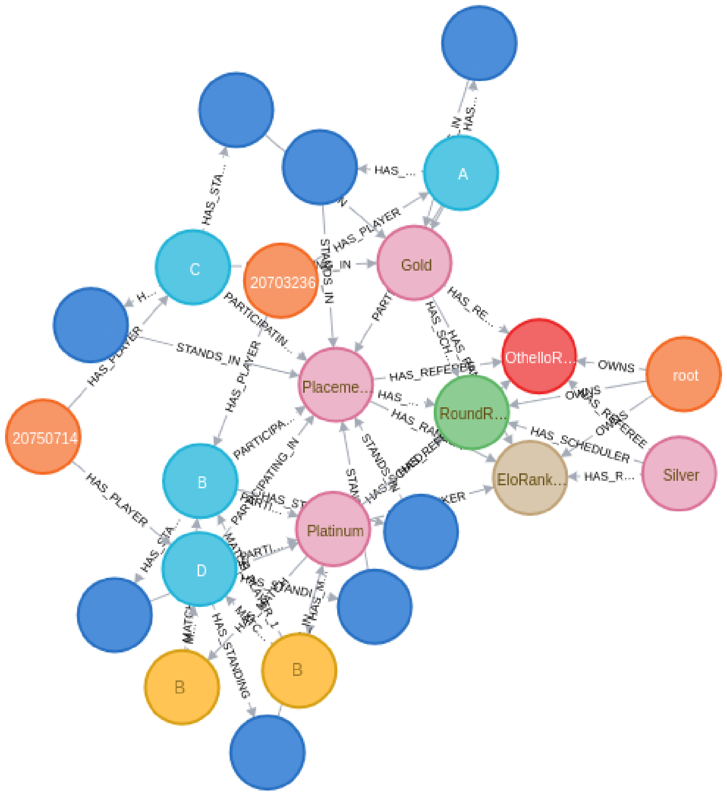


Figure 7.10: An example of the data connected in the Neo4j database

Bibliography

[1] Masters Traditional Games. *Rules and Instructions for Reversi and Othello.* 2019. URL: https://www.mastersofgames.com/rules/reversi-othello-rules.htm.

[2] Google Inc. *The Go Project.* Aug. 2020. URL: https://golang.org/project/.

[3] Adam Newell. *What is Elo? An explanation of competitive gaming's hidden rating system.* Jan. 2018. URL: https://dotesports.com/general/news/elo-ratings-explained-20565.