









# SOFT 355 Report

## Functionality

The application I have made is an online scrabble game that utilises using web sockets to create a room-based system. The room system allows for multiple games to be played simultaneously. The game of scrabble is fully functional except for the exclusion of blank tiles. This means that moves are checked, processed and points are calculated automatically and awarded to the right player.

The initial flow of the application is that the player must initially create an account on the login page or sign in to an existing one.





### Sign Up

	<input type="text" value="Login ID"/>	
	<input type="text" value="Ingame Name"/>	
	<input type="password" value="password"/>	
	<input type="password" value="verify password"/>	

Create Account

Login

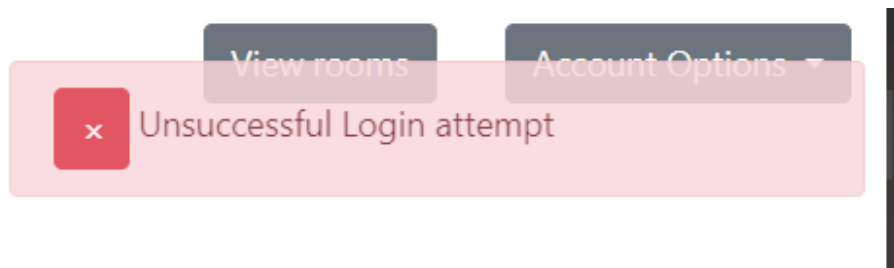
### Login

	<input type="text" value="test"/>	
	<input type="password" value="...."/>	

Play

Sign Up

If the user attempts to login to an account that doesn't exist or has incorrect details, it should show a toast notification:



Once logged in, the user is taken to the room view component. Here they can join a room or create a room by naming it and pressing the create room button.

View rooms

Account Options ▾

**test**

Players in game:

While in the lobby, other players should be able to see you as a player in the game.

Room name

Create Room

**test**

**Players in game:**

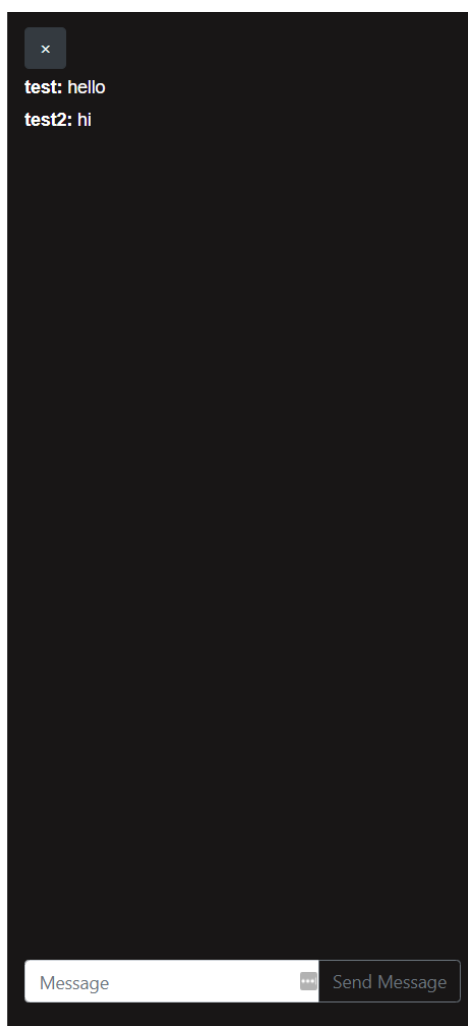
test

Join Room

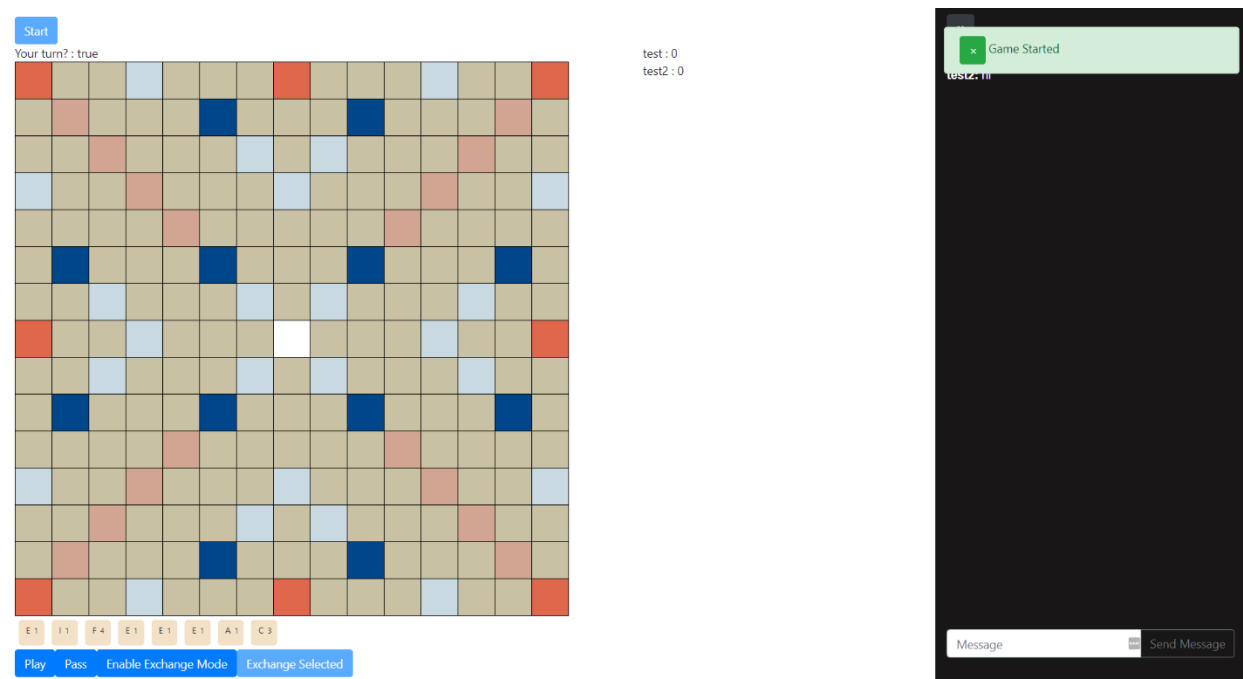
Once in a room, players can chat using the chat bar, this is toggled into view by clicking the chat button in the top right.



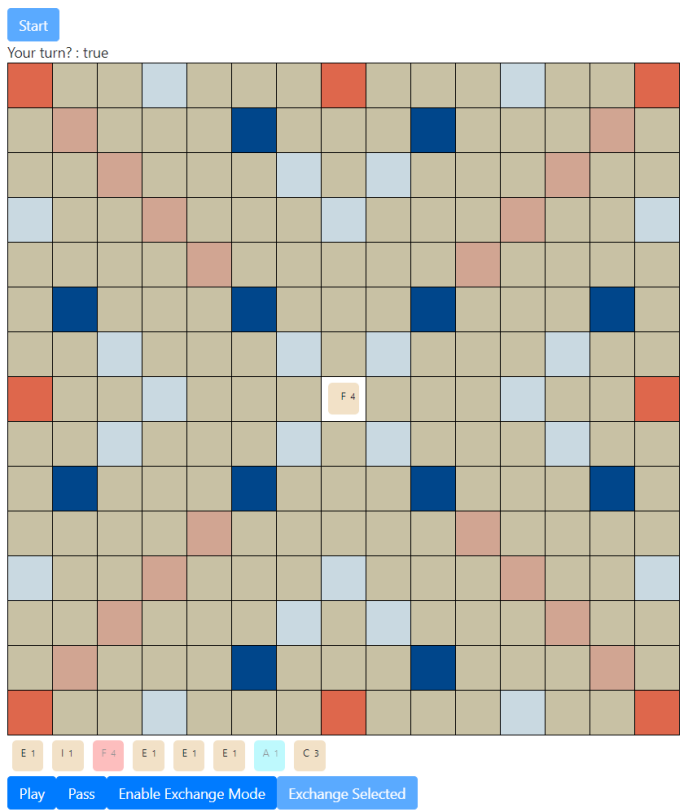
Once the chat bar has opened, you can send messages to other players.



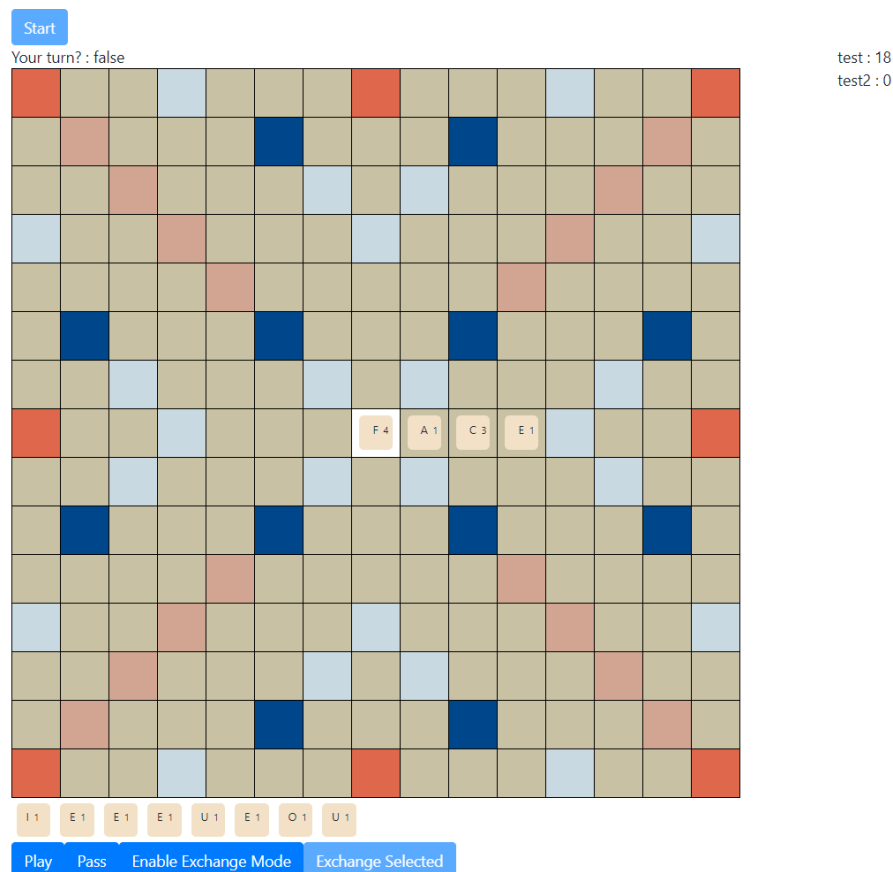
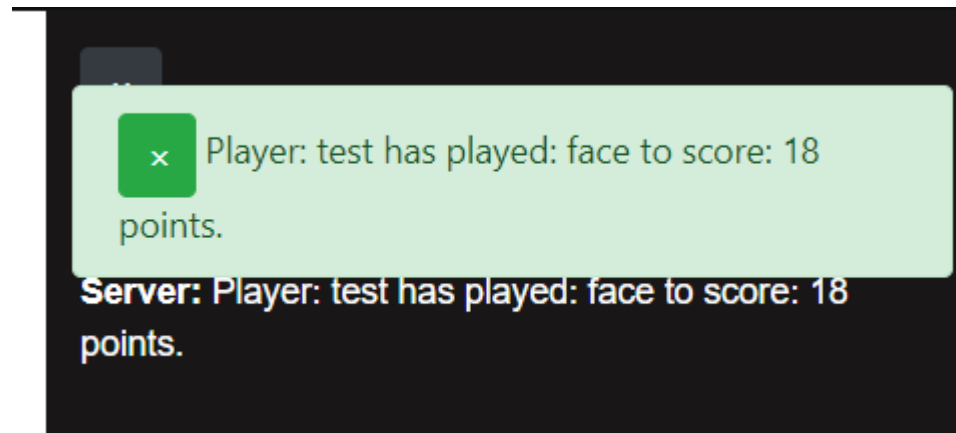
Once as many players as you wish join, you can start the game by clicking the start button.



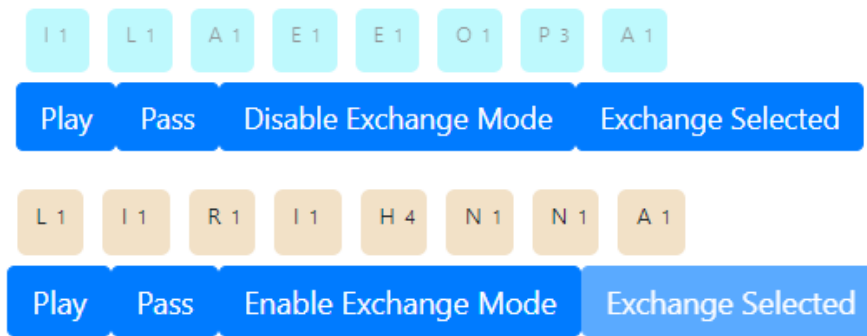
This will then show the games board and your hand. You can select a tile and it will turn blue. Afterwards you can select a position on the board to place it, once on the board it should turn red, indicating it can't be used again.



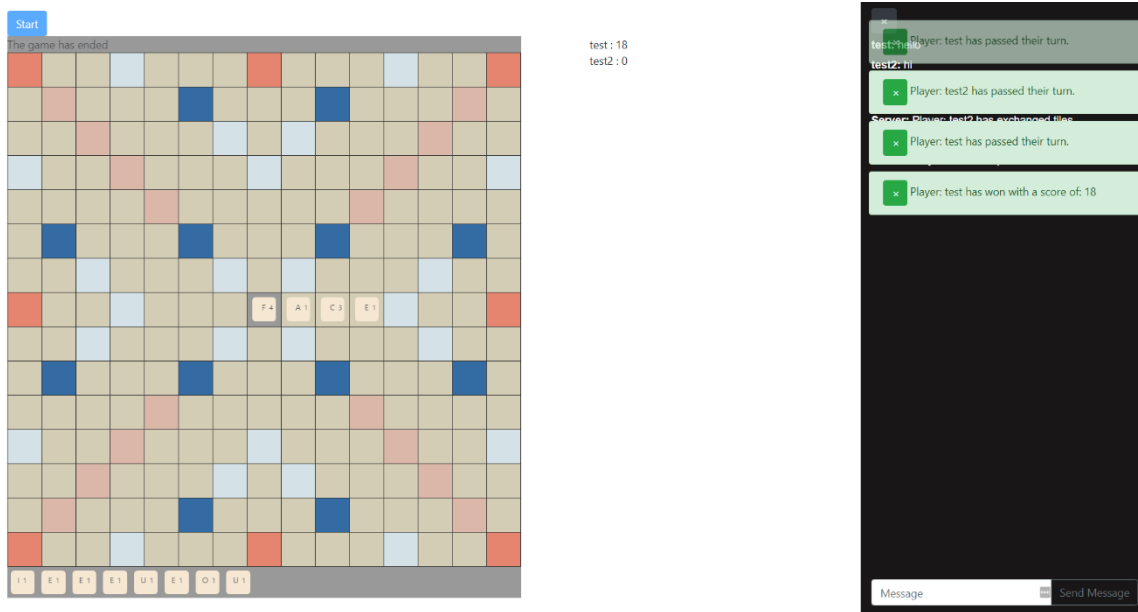
If you wish to pick up the piece, you can re-click it on the board to pick it up. Once all pieces are in place, you can then click play to make your turn. If all is well, a successful toast should show, and a message is placed in chat in case you miss the toast.



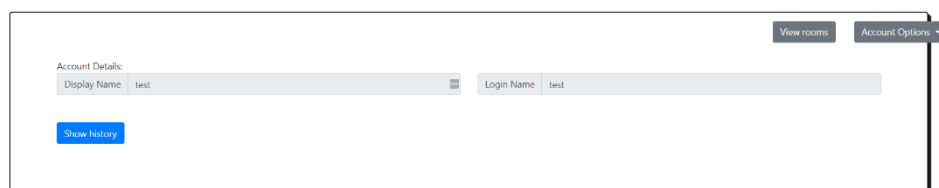
You are also able to pass your turn or exchange tiles. Exchanging tiles is done by enabling exchange mode and clicking on multiple tiles. You then click on the button and it will exchange those tiles for you.



Once a player has passed twice in a row, the game shall end, and everything should be unclickable.



After a game has taken place, under the account dropdown you are able to view your account.



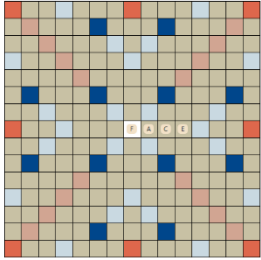
Here there is a section that shows recent game history.

Hide History

Match History:

Match: test

Date: Jan 8, 2020, 8:10:39 PM



Scoreboard

test : 18

test2 : 0

Turn History:


Player: test has played: F, A, C, E

Player: test2 has exchanged 8 tiles.

Player: test has passed their turn.

Match: test2

Date: Jan 8, 2020, 2:40:13 PM



Scoreboard

The technologies used were:

Angular + tslib + zone.js : This was used over angular.js as I was more familiar with it from my year in placement. [1]

Chai/mocha: For creating testing suites for testing the server-side game code. [2]

Express: This was used to expose API methods for dealing with the creation of players and rooms [3]

Jquery and Bootstrap: Used to style the website. [4][5]

Karma/mocha-junit-report: This was used to produce j-unit style xml files that could be used within Jenkins to display test results. [6][7]

Js-sha256: Used for hashing passwords for accounts [8]

Mongoose + mongodb: The database and service used to interact with. These were used with express to create my API methods. [9][10]

Socket.io: Used for communicating between my client and server. [11]



## Requirements

The application is aimed at people who have a rough idea of how online games work. People who have used sites like skribbl.io (online Pictionary) or AMQ (an online song recognition game), would be familiar with the layout and functionality of the site. The style and gameplay of the game was made to mimic scrabble, so it would also be appealing to fans of the real boardgame too.

The features that were included were:

An account system: This is used to identify players and to allow for game history to be kept. This was added so players can look back at games and see how they panned out. The account system uses sha256 hashing to prevent passwords being stored as plain text in the database.

A room-based system: This was done to allow for multiple games to be played at once. This prevents the server from being locked out from other users if 1 set is already playing.

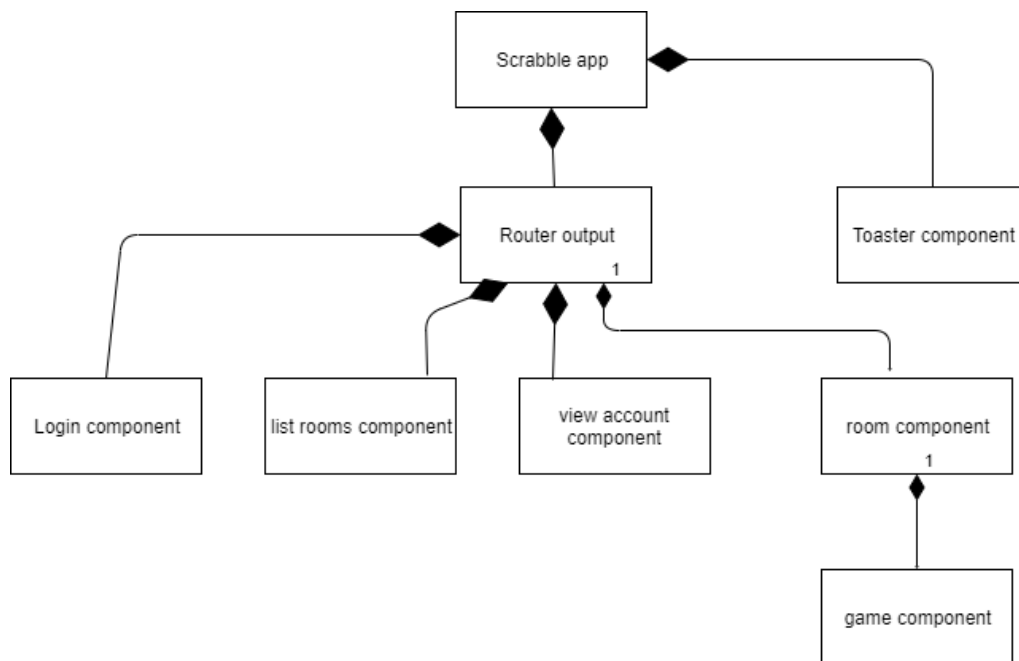
A chat system: The chat system was initially added as an aid to learn how to use socket.io. This was kept in to allow people from playing games to communicate without needing to use external services.

A scrabble engine: The scrabble engine was created to allow players to play scrabble without needing to make the player determine any scores or if a move is valid.

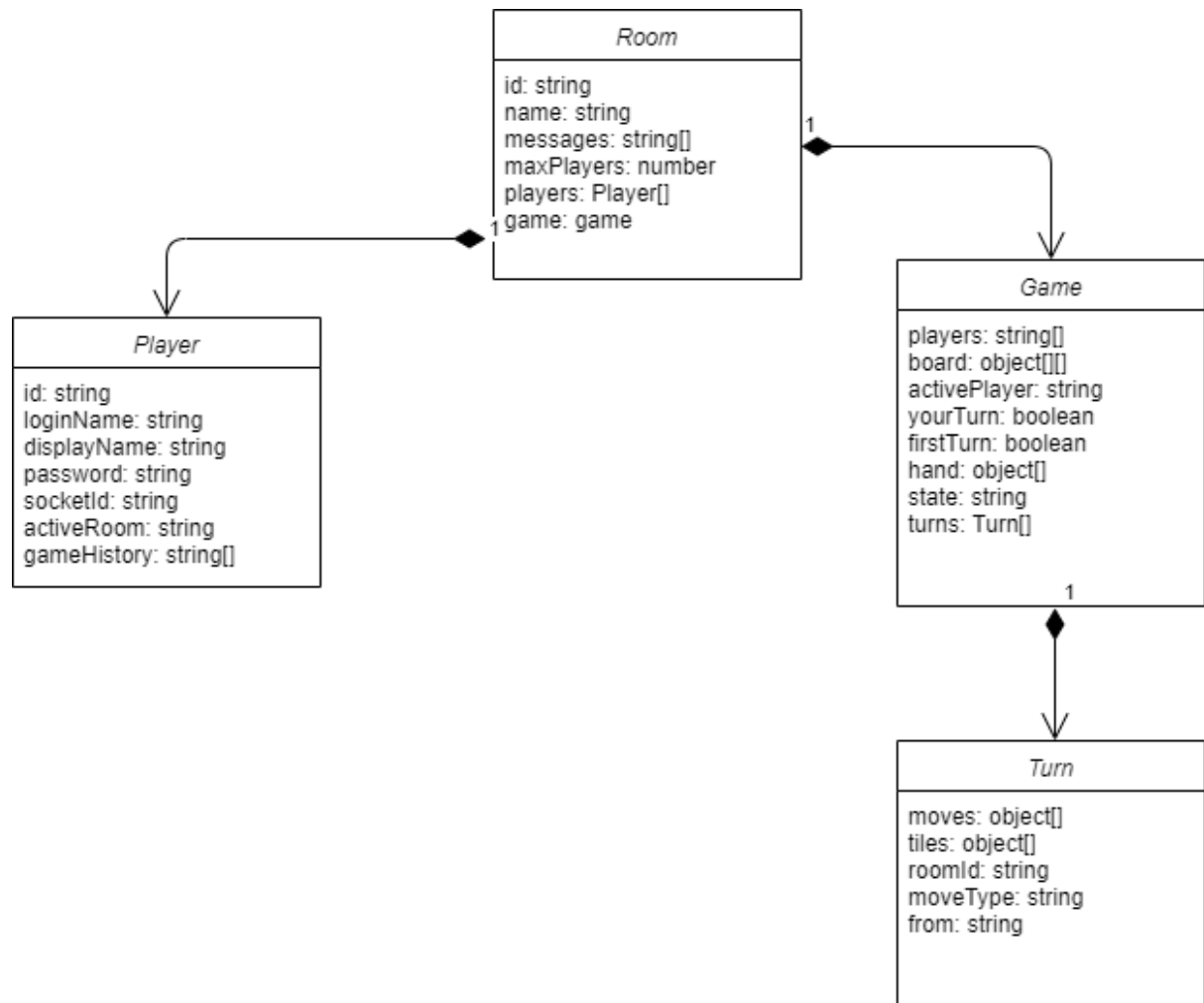
## Design

The system architecture that I have decided to use is client server. The client is created using Angular and the server uses a Node core with express to serve API methods. The server also uses mongo as a non-SQL backed database to store player accounts and past games.

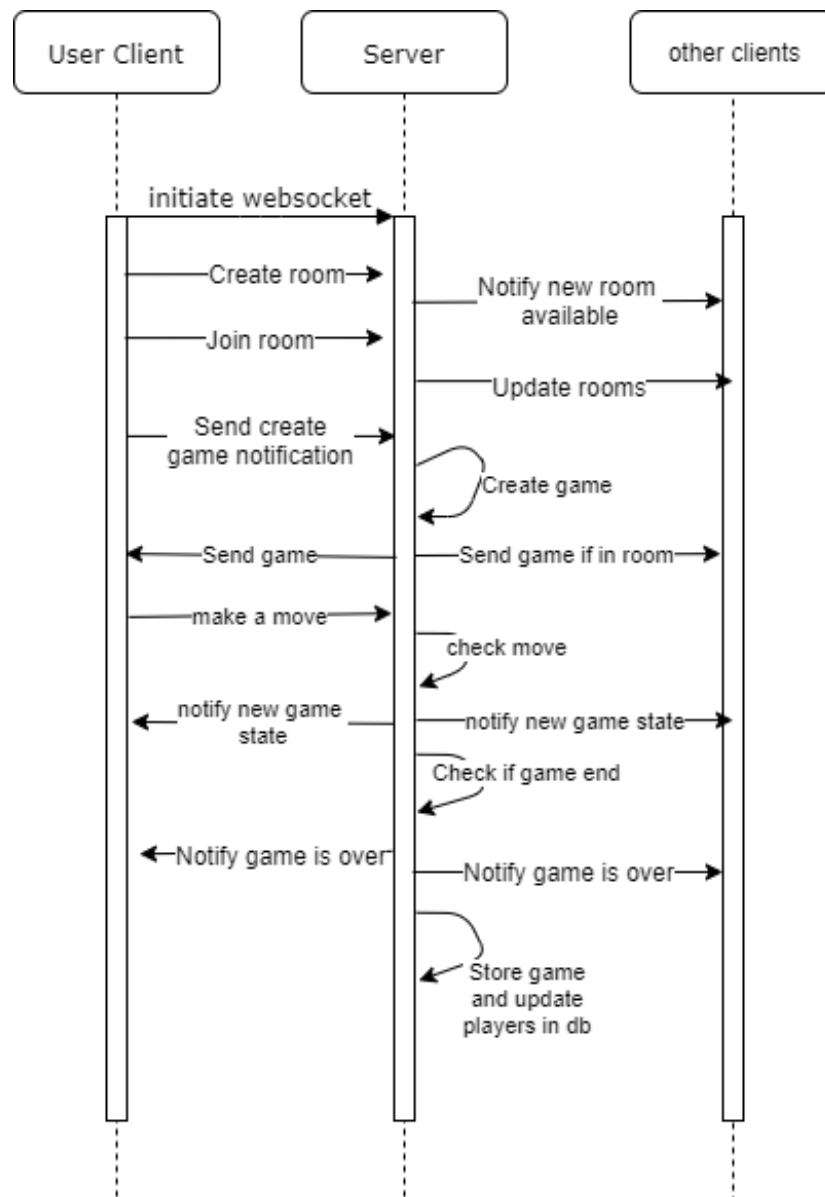
One advantage of using angular was that the client is structured in a way where each component is organised separately. This means it's easy to keep code separated. This is the breakdown of the angular components.



The server was used to manage the game and the game logic. Here are how the game objects were composed.



This would be a standard flow of what would happen when the system is being used after being logged in.



The objects that were stored within the database were minimalised versions of objects used by the server. The game object had a lot of properties removed from it for efficient storage, for example:  
There would be no need to store the end game state if we have the moves that created it.

Player
playerId: String loginName: String playerName: String password: String socketId: String activeRoomId: String gameHistory: [String]

Game
id: String maxPlayers: Number name: String players: [String] createDate: String game: String

## Testing

For testing I used a combination of karma and mocha/chai. By default, Angular creates a testing file for each one of the components that is generated. Within these I created some tests to make sure correct methods were called and classes were assigned when certain values were met.

For server testing, I created an API test suite that ensured the API methods worked as expected. I also created an extensive testing file (/server/game/test.js) to ensure that my scrabble engine worked efficiently. Whenever a bug would show up, I would write an additional test to prevent the issues from showing up again if I changed the code.

For both testing suites, I installed a library that allowed me to output j-unit test result files which are xml files that are produced containing the results of the test. These were useful as Jenkins has a deployment step where you can search for these files and publish them as test results.

On Jenkins I set it up so that the project would build automatically every single day at 3pm. This meant I was able to keep on top of the tests to ensure that everything was running ok. The configuration used is down below:

## Source Code Management

- ☐ None
- ☒ Git

### Repositories

Repository URL

Credentials

 Add

Advanced...

Add Repository

### Branches to build

Branch Specifier (blank for 'any')

Add Branch

### Repository browser

(Auto)

### Additional Behaviours

Add

- ☐ Subversion

## Build Triggers

- ☐ Trigger builds remotely (e.g., from scripts)
- ☐ Build after other projects are built
- ☒ Build periodically

### Schedule

H 15 \* \* \*

Would last have run at Wednesday, 8 January 2020 15:04:50 o'clock GMT; would next run at Thursday, 9 January 2020 15:04:50 o'clock GMT.

- ☐ GitHub hook trigger for GITScm polling
- ☐ Poll SCM

### Build

Execute Windows batch command

Command


```
npm install  
npm test
```

See [the list of available environment variables](#)

Advanced...

Add build step ▾

### Post-build Actions



**Publish JUnit test result report**

X

?

Test report XMLs

[Fileset 'includes'](#) setting that specifies the generated raw XML report files, such as 'myproject/target/test-reports/\*.xml'. Basedir of the fileset is [the workspace root](#).

☐ Retain long standard output/error

Health report amplification factor

1% failing tests scores as 99% health. 5% failing tests scores as 95% health

Allow empty results

☐ Do not fail the build on empty test results

?

Add post-build action

▼

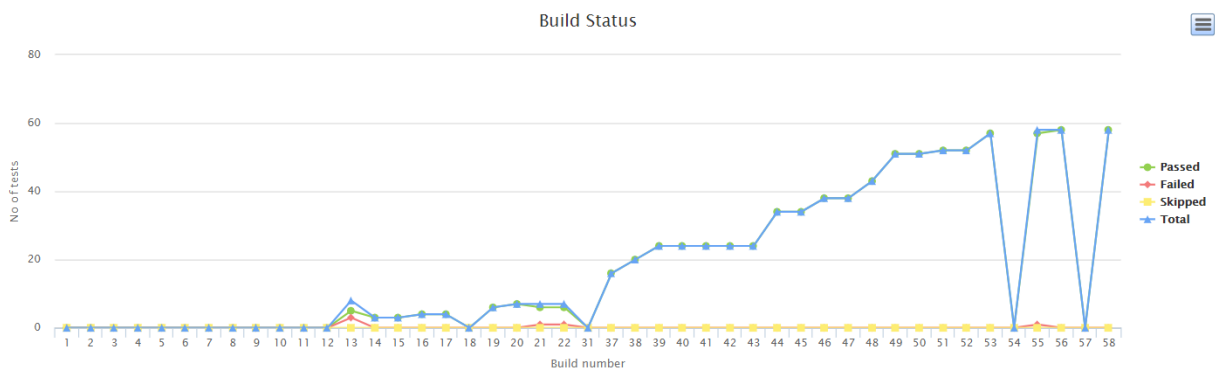
This strategy was effective as it meant that I was able to find broken tests without having to spend time running the tests and waiting for them to finish.

For usability testing, I set up my application to be deployed on Heroku. This meant that I was able to allow other people to connect to and use my application. Having users test my game was a useful experience as they were able to discover bugs/oversights that I might have overseen. A couple of bugs that came out of this was that players weren't leaving when the tab was closed. Initially I used the `onDestroy` functionality, but this is only when the component closes. I had to then investigate events on when the tab is closed to solve this. This approach to testing allowed me to find things that I would have easily overlooked. For example, when you design and make something, you tend to use the app how you designed, which isn't necessarily how external users will try and use it.

## DevOps pipeline

While developing the application, my main IDE was Visual Studio Code. This is because the linter and features that it provides are better than any of the alternative programs that I have used. As a testing tool I used chrome and the chrome debugger extension. This allowed me to place breakpoints in VS code without having to open the chrome developer console and having to locate the JavaScript that I was testing. For testing I have also tried to use a variety of devices, such as my laptop, pc and mobile phone to see how the website loads/works on them. Not as much time has gone into testing on other devices so If I were to redo the project again, I would like to spend more time on focusing that it loads properly and in a way that is usable on mobile devices.

As mentioned in previous section, I was able to set up the Jenkins automated build tool to allow for continuous testing and project building. This saved a lot of time as I didn't have to go and build the project and run the tests every single day. It would have been better if I was able to setup builds to run when the GitHub project was updated, however I was unable to do this as my Jenkins was set up locally and my current living situation doesn't allow me to set up my internet connection to allow for incoming connections. Using Jenkins also allowed me to generate a graph over time of test results:



The occasional 0 is where the project was misconfigured or when it was unable to communicate with GitHub. But the graph shows a trend of over time the number of tests increasing showing test-driven development.

Similarly, to Jenkins, I used Heroku to deploy my application online. This allowed me to investigate different bugs and to see what the process might be like if I were to do this to another project. Although it wasn't part of the specification, I am glad I still did this as the additional time my friends spent bug finding and testing allowed for the project to become more stable over time. This however wasn't done as continuously like Jenkins but was done at more incremental steps when I felt the project had a significant enough of progress made from the last deployment.



## Personal reflection

Reflecting on the project, I am glad that I spent the time setting up Jenkins and Heroku for deploying my application. It had allowed me to use my time more effectively working on the application. However, some time was wasted initially in setting these up, but I feel like the positives it brought outweighed the initial teething issues.

I am also glad that I decided to use Angular and not the angular.js library. The local angular component generation saves a lot of time creating components instead of having to create the 3 independent files locally in a folder. Angular also creates test files for each component and service so it also makes it easier to test each component.

One of the things I will take away from this project is the process of approaching logical problems. A board game such as scrabble is a very good test as there are set rules and processes you should abide by to emulate the real board game. Developing my scrabble engine posed quite a few challenges such as how to detect if a move is valid. It posed a good technical challenge and taught me that approaching a problem by attempting to solve a simpler version of the same problem is a valuable lesson.

## Appendences:

- 1: <https://angular.io/>
- 2: <https://www.chaijs.com/>
- 3: <https://mochajs.org/>
- 4: <https://jquery.com/>
- 5: <https://getbootstrap.com/>
- 6: <https://github.com/karma-runner/karma-junit-reporter>
- 7: <https://www.npmjs.com/package/mocha-junit-reporter>
- 8: <https://www.npmjs.com/package/js-sha256>
- 9: <https://mongoosejs.com/>
- 10: <https://www.mongodb.com/>
- 11: <https://socket.io/>