Tug of Words

Regression and Incremental Testing Log - Sprint 2

Jenna Ellis | Charlie Crouse | Jiwon “Daniel” Kim | Anoop Jain | Prashanth Koushik | Ammar Husain

---

Classification of Components

Server Components

App

- Holds all of the server configurations/initializations. This is where express and socket.io are initialized as well as where all of the modules/routes are connected and exposed to the server.
- Inputs
    - *Port* {Number}: the port on which to run the server. For development we default this port to 8000, but in production it is a hidden environment variable.
- Outputs:
    - *Server* {Object}: the initialized server, which is outputted so it can be initialized and run in different environments (it is helpful to run it in a isolated testing environment)
- Depends on:
    - **User**, **Lobby**, and **Game**
- Used In:
    - **App** tests

Fire

- Module that holds the config and the initialization of the connection to our real-time data storage, firebase.
- Inputs
    - *Api Key* {String}: the secret api key used to verify our connection
    - *Auth Domain* {String}: the domain of our app
- Outputs
    - *Firebase.db* {Object}: this is the persistent connection to our data store that allows us to watch out for and make changes in real-time
- Used In:
    - **User**, **Lobby**, and **Game**

User

- Module that holds all logic central to users (i.e. creating a user or removing a user). Routes that operate on user data are defined here and are exposed by the **App** module to allow for CRUD operations through HTTP requests.
- Inputs:
    - *User ID* {String}: this is a unique string that is used to identify our temporary users in our data store.
    - *Username* {String}: this is a name that a user chooses to be publicly identified by in game lobbies or in a game.
- Outputs:
    - Several functions that provide containerized logic on users (e.g. createUser and removeUser)
    - *User Router* {Object}: this is the router that will be utilized by the **App** module to expose the user endpoints
- Depends on:
    - **Fire**
- Used In:
    - **App, User** tests

Lobby

- Module that holds all logic central to lobbies (i.e. creating a lobby or removing a lobby). Routes that operate on lobby data are defined here and are exposed by the **App** module to allow for CRUD operations through HTTP requests.
- Inputs:
    - *Lobby ID* {String}: this is a unique string that is used to identify our temporary users in our data store.
- Outputs:
    - Functions that provide containerized logic on rooms (e.g. createLobby and removeLobby)
    - *Lobby Router* {Object}: this is the router that is utilized by the **App** module to expose the room endpoints
- Depends on:
    - **Fire**
- Used In:
    - **App, Lobby** tests

Game

- Module that holds all logic central to the game. All actions that if potentially tweaked by the client would fail the game are defined in this module. Although it adds some bulk to the server, this is added security and is an easy way to prevent users from trying to manipulate or hack their way into winning (i.e. generation of words, validation of words, etc.)
- Outputs:
    - Helper functions that hold the logic for in game actions that all users will continuously see (i.e. adding points, removing points, generating random words).
- Depends on:
    - **Fire**
- Used In:
    - **App**, **Game** tests

Client Components

App

- Module that acts as a parent container for the entire client. This is where routing is done (i.e. where the components are bound to specific routes based on the current state of the app)
- Inputs:
    - *Auto User Creation* {function}: check the browser session history to automatically remember and assign the user their previous username and credentials.
    - *User ID* {String}: the current user id bound to the session (null if this is a new user)
- Outputs:
    - The app component connected to the global reducer- the object that manages the state of the client
    - Unconnected app component for testing
- Depends On:
    - **<MainMenu>, <Lobby>, <Game>**

Main Menu

- Module that allows users to create a temporary user and then join a random lobby to play or create a private lobby that will allow them to play with people who possess an invitation.
- Inputs:
    - *Username* {String}: the name by which the user will be publicly identified
    - *Game Type* {Action}: this is determined by whether the user hits a button that says `Play Now` or `Create Private Lobby` which will then spawn a sequence of events corresponding to their given choice.
- Outputs:
    - The main menu component connected to the global reducer

    - Unconnected main menu component for testing
    - *Game Sequence* {Event}: this is again determined by whether the user chose `Play Now` or `Create Private Lobby`. If `Play Now` was chosen then the user will be sent to best fit game lobby that our server has found for the user, or if `Create Private Room` was chosen then the user will be shown a confirmation dialog that if confirmed will send the server a request to create a new private lobby, then put the user in the empty private lobby.
- Depends On:
    - ***<Text Input>***: stateless component that renders a textual input field for the user
    - ***<NewLobbyDialog>***: component that renders a confirmation dialog when the user requests to create a private room
- Used In:
    - **<App>, <MainMenu>** tests

Lobby

- Module that will give the users a preview of the other users and will display which users are on which team.
- Inputs:
    - *Team 1 Members* {Array}: this is a list of the usernames, user ids, and statuses that belong to the members on the first team. It will be updated and re-input anytime a new user joins the lobby/team.
    - *Team 2 Members* {Array}: this is a list of the usernames, user ids, and statuses that belong to the members on the second team. It will be updated and re-input anytime a new user joins the lobby/team.
    - *Team Assignment* {Action}: if the room is public, then this action will be defined by the client which will place them on the team with the least amount of users, but if the room is private then the user will define the action by clicking a join team X button that will define an action to assign the user to team X.
- Outputs:
    - Lobby component connected to the global reducer
    - Unconnected Lobby component for testing
    - The final team assignments when the game start sequence occurs
- Used In:
    - **<App>, <Lobby>** tests

Game

- Module that will handle the rendering the state of the game to all of the users in a given room.
- Inputs:
    - *Team Assignments* {Object}: the team assignment info that includes the user IDs and names for each user in the lobby.
    - *Target Words* {Array}: array of length N if there are N users in the lobby. This is essentially the mapping of a user to a word that they need to type correctly.

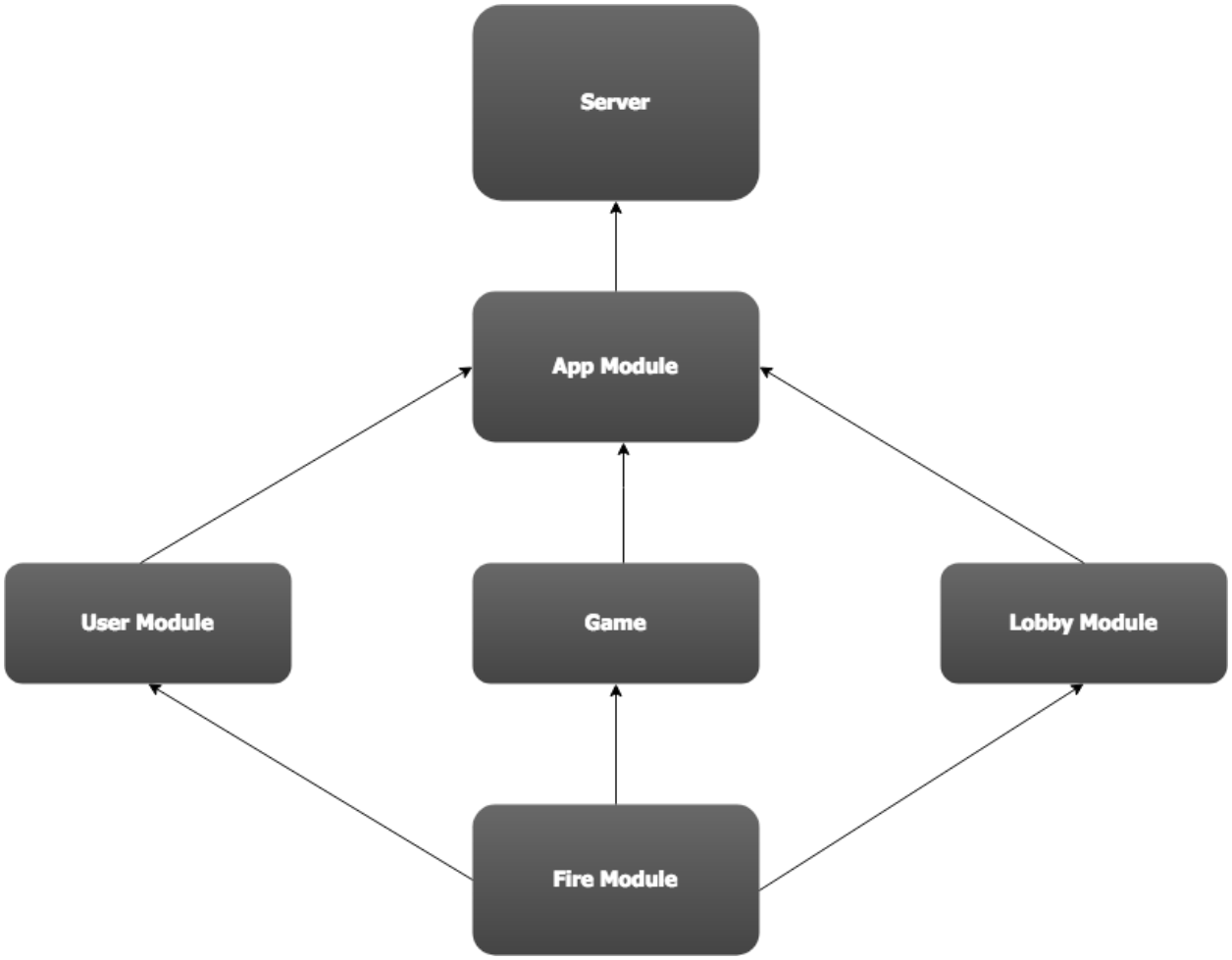
When a user is verified to have typed a word correctly, then a new word from the server is mapped to the user.

- Outputs:
    - (In-game):
    - The word that each user has to type correctly in order to score points or pull the rope for their team
    - When a word is submitted (by pressing enter or space) the status of whether or not the word is correct is displayed
    - The current game status i.e. which team is winning and by how much
    - (Post-game):
    - The results of the game i.e. the winning team and which users contributed the most points to the team.
    - (Component)
    - The game component connected to the global store
    - Unconnected game component for testing
- Used In:
    - **<App>, <Game>** tests

Testing Techniques

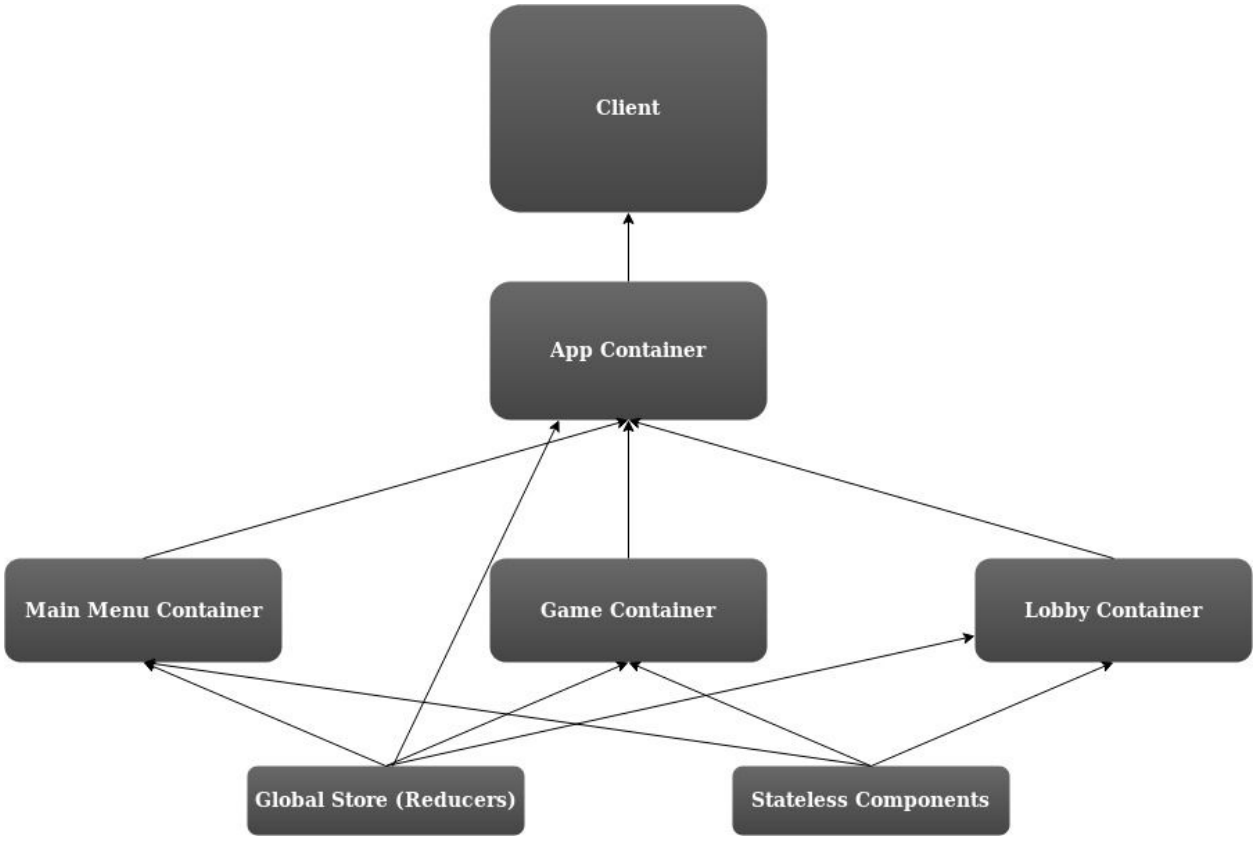
Server

It was more beneficial for us to begin testing our server modules with a bottom-up approach. The drivers required are rather simple to code, e.g. some input that is received from the client is much easier to code than a stub for the higher order server module. Additionally the output from testing the components in isolation as submodules, then add tests to simulate the conditions of a main module.



Client

Similarly to the server, the client is being tested with a bottom-up approach. The nature of react is to develop UI components as single units, that way they can be reused without having to repeat code. This also helps us test the components as submodules, which are then get added to a main module as the components are used in larger containers. The drivers might manifest itself as a function that would be given as input to a component, but due the nature of bottom-up testing, we do not have access to that function from an outside module, so a driver is needed to imitate that input.



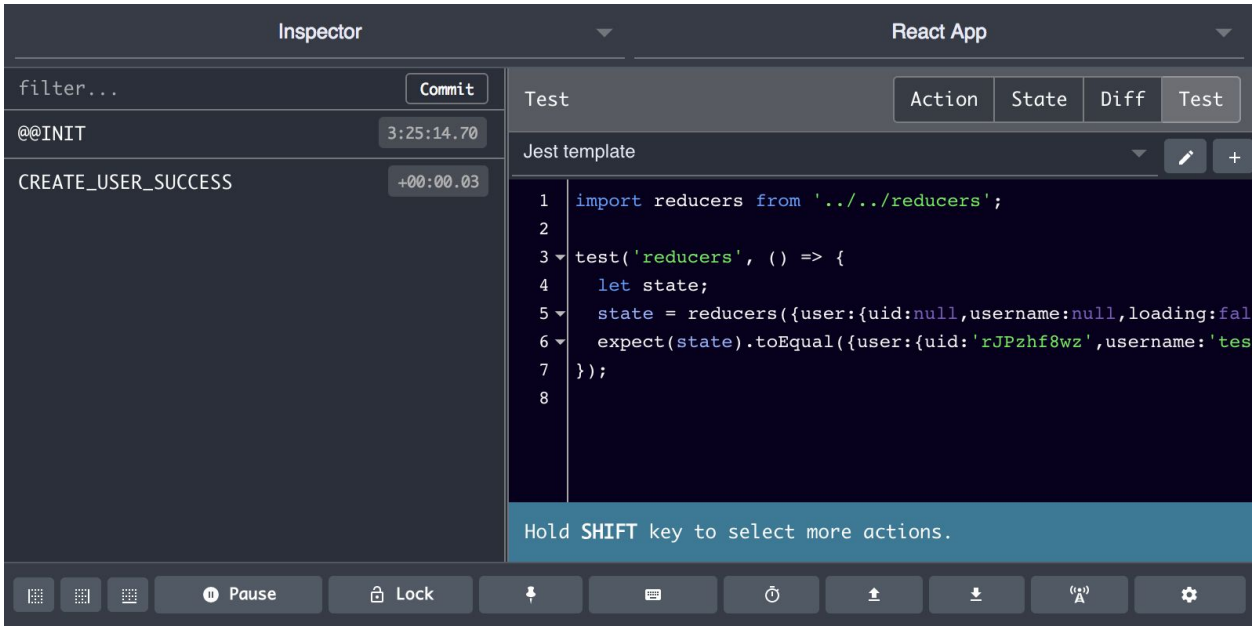
Testing Automation

Server

The server implements its test with Mocha JS. This is an automated testing framework that allows us to define `Suites` that encompass all of the tests for a specific submodule. Both our server and our client have package.json files that define metadata about the project, show the dependencies, and also includes some runnable scripts. We have included a script that enables us to automatically run all of our suites or specific suites and even individual test cases.

Client

React comes built with a testing framework named Jest. It is very similar to Mocha JS in syntax and in style. Similar to the server, there is a built in script that enables us to run all of our testing suites or just the tests for individual components. In order to test components in isolation, we have added a library named enzyme that allows us to test UI components in a shallow testing environment, which makes bottom-up testing very easy to implement. In order to test the global store (redux) we are utilizing the power of Redux Developer Tools to automatically generate test cases that capture a specific state of our client, and make logical assertions on that. For example:
Here is a test case generated by the tool that can assert that the background state of our client is exactly what it is supposed to be while on the main menu:



This is very useful for regression testing when we add more complex features that could tweak the state of our client in ways that we haven't thought about.

Defect Log

Server

Module	**Fire**		
Defect #	Description	Severity	Solution
1	Firebase snapshot values seemed to not work correctly, returning null values instead of real values in the database.	2	The problem was a schema-related issue. Fixing the schema to stay consistent throughout the code solved the problem.
2	During testing, watching the Firebase console was not helpful because the values were not updating correctly in the console when Firebase objects were being deleted and created even through the tests were working.	1	Restarting the computer solved the problem.

Module	**User**		
Defect #	Description	Severity	Solution
1	N/A		

Module	**Lobby**		
Defect #	Description	Severity	Solution
1	The name of the method that would let a user leave a lobby was changed from removeLobby() to leaveLobby(). This was not changed in the lobby.test file, so the test was failing. This was very slightly confusing to the person tasked with looking at the reason for the failed tests.	1	The test writer discovered that while the functionality of the method did not change, it was required that they change all instances of the removeLobby() to instances of the leaveLobby() method.
2	There was no hook for joining an open lobby through the URL given, meaning that an unlimited number of users could join any lobby.	2	The hook for joining a lobby was written where if the number of users is greater or equal to 50, the user cannot join that lobby.

Module	**Game**		
Defect #	Description	Severity	Solution
1	Firebase would not update quickly enough for strictly synchronous addPoint and removePoint functionality. This means that the method would return before the firebase operation completed.	2	By using the “await” and “async” keywords in JavaScript, the methods that use firebase will wait for the firebase operation to complete before moving on.

2	An initial design decision that was made was to include the socket.emit() functions in the game.js file rather than the overarching app.js file. This was later considered bad coding practice.	1	All of the socket.emit() were moved into the app.js file from the game.js file. The methods in the game.js file now end with “return” calls. This also made testing for these methods much easier.

Client

Module	**App**		
Defect #	Description	Severity	Solution
1	N/A		

Module	**MainMenu**		
Defect #	Description	Severity	Solution
1	N/A		

Module	**Lobby**		
Defect #	Description	Severity	Solution
1	In the leaveLobby method, the user was removed from the team, but not from the users list associated with the lobby.	1	Remove user from users list in the leaveLobby method.
2	Link sharing was not being dynamically updated properly	2	Fixed modal implementation in which url is accessed

	upon first implementation.		through JS and not through routing local to the application.
3	The client could not connect to the socket when joining a public room because the socket connection requires a lobby ID which is not generated until user attempts to join a public lobby.	3	Created a server call that returns the public lobby that is currently being populated which is used by the client to retrieve the lobby ID used for the socket connection.

Updates to Product Backlog:

The following is the updated version of our product backlog, in which completed stories, stories that are in-progress, and stories planned for the next sprint are identified:

Id	Functional Requirements	Status
1	As a user, I would like to create a private room	Completed in sprint 1
2	As a user, I would like to join a random public room	Completed in sprint 2
3	As a user, I would like to join a private room	Completed in sprint 1
4	As a user, I would like to automatically be assigned to a team in a public room	Completed in sprint 2
5	As a user, I would like to earn points for my team for correctly typing a word	Completed in sprint 1
6	As a user, I would like to see my individual points change as I play the game.	Completed in sprint 2
7	As a user, I would like to see my individual points and teams points total after I complete a game.	Completed in sprint 2
8	As a user, I would like to see statistics on my typing speed in a given gamen (Correct words typed per minute).	Completed in sprint 2
9	As a user, I would like to see my opponents lose points for typing a word incorrectly.	Completed in sprint 2
10	As a user, I would like to pick which team to join in a private room.	Completed in sprint 1
11	As a user, I would like to share a link to my private lobby with friends.	Completed in sprint 2
12	As a user, I would like to be identified with a game tag within my room (no account creation)	Completed in sprint 1
13	As a user, I would like to be on teams that may not be equal in size in a private game.	Completed in sprint 1
14	As a user, I would like the game to start a predetermined	Completed in

	amount of time after the first two users join a private lobby	sprint 2
15	As a user, I would like to see the “rope” move towards my team’s side when we are winning.	Removed from backlog - not implemented
16	As a user, I would like to see the rope move towards my opponent when I make typing mistakes.	Removed from backlog - not implemented
17	As a user, I would like the words generated by the game to be sufficiently random.	Completed in sprint 1
18	As a user, I would like to get “hot bonuses” when I have a streak of correctly typed words.	Removed from backlog - not implemented
19	As a user, I would like to access Tug of Words via a public URL.	Completed Sprint 2
20	As a user, I would like to see the correctly typed words of my team displayed on the enemies’ screen.	Removed from backlog - not implemented
21	As a user, I would like to be on teams of relatively equal sizes when playing in a public room	Completed in sprint 2