Rappakalja - Project Report

Theo Wiik, Emanuel Enberg, Jonathan Heinin, Jesper Larsson

March 2021

Contents

1 Introduction **3**
1.1 The group . . . 3

2 List of Use Cases **4**

3 User Manual **5**

4 Design **9**
4.1 Frontend . . . 9
4.1.1 Connection with Websocket . . . 9
4.1.2 The useGame hook . . . 9
4.1.3 The Game View . . . 10
4.1.4 Mutating and Creating Data . . . 10
4.1.5 The Modify View . . . 11
4.2 Backend . . . 11
4.2.1 The Controllers . . . 11
4.2.2 CRUD-actions from the Frontend . . . 11
4.2.3 The Model . . . 13
4.2.4 Instantiating the Websocket . . . 13

5 Application Flow **14**
5.1 Joining a Game . . . 14
5.2 Submitting an explanation . . . 15

6 Package Diagram **16**

7 Code Coverage Report **17**

8 Model Diagram **18**

9 Responsibilities **19**
9.1 Theo Wiik aka backend wizard . . . 19
9.2 Emanuel Enberg aka best practice instructorboi . . . 19

9.3 Jonathan Heinin aka frontend dev guru 19
9.4 Jesper Larsson aka the design god 19

1 Introduction

Rappakalja is an online game accepting up to 10 players. The game starts off with the players receiving a difficult and unusual word. Their goal is to try to describe this word as convincing as they can to trick the other players in the session that their explanation of the word is correct. After everyone has written their explanation, a screen with all the explanations pops up including the correct description of the word. The players then vote on the description they think is correct. Picking the correct answer or making the other players pick your description yields points.

Link to the git repo: `https://github.com/theowiik/word-game`

1.1 The group

Figure 1: This is us, the group behind the project

2 List of Use Cases

- Select settings for a game
- Creating a game
- Joining a game
- Allow other players to join the game
- Start a game
- Entering and submitting an input
- Selecting an input
- Add a new category to your game
- Add a new word to your game
- Modify words in your game
- Delete words from your game
- See the leading player
- See who won the game as well as all points of the players
- See who chose what answer
- See who wrote the answer

3 User Manual

To play the game, visit `http://w-g.herokuapp.com`. There you can choose to either start a new game or join an existing one.

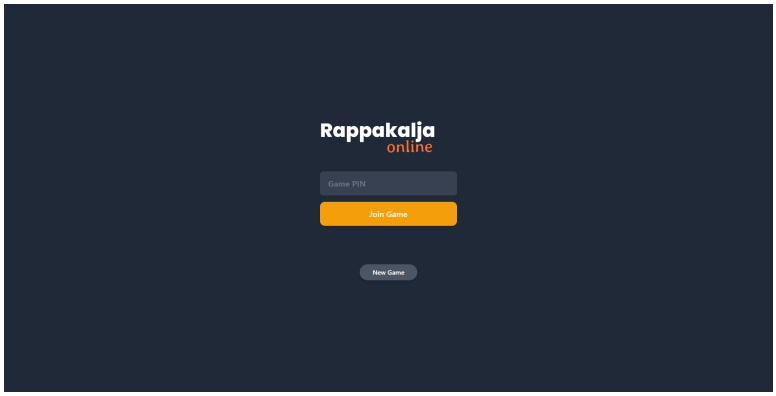

Figure 2: Landing page with a field for pin code to join a game

Choosing to start a new game you can pick between different categories of words to include in your game. For example you could choose swedish or english words as seen in the picture below.

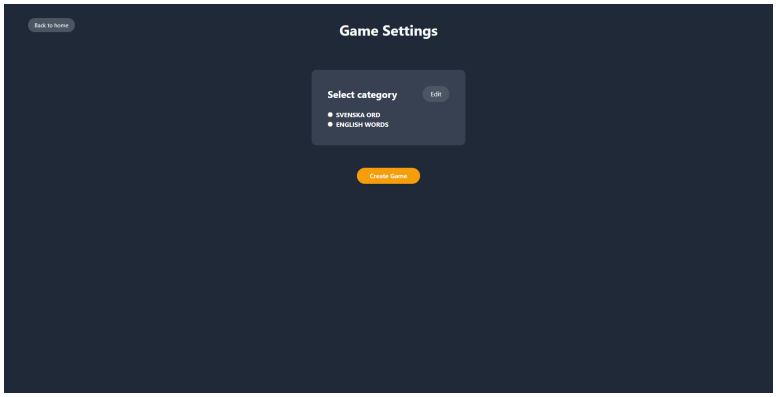

Figure 3: The page for creating a new game

There is also an option to add your own words and categories into the game by pressing the "Edit" button which brings you to the page as seen in the figure below.

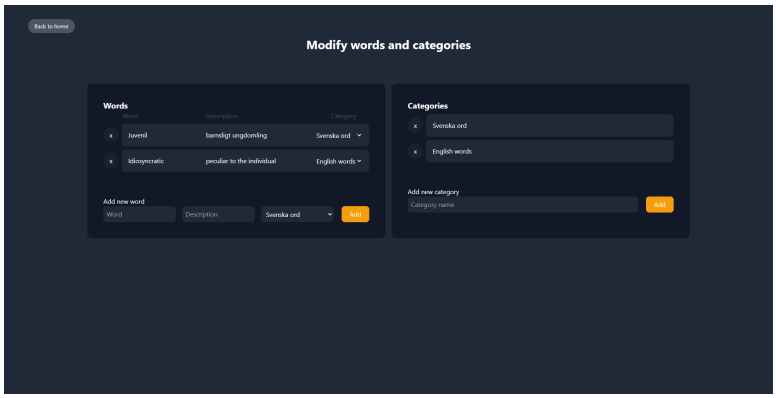

Figure 4: The screen for modifying the available categories and words

After this you press start game and you are redirected to a lobby where you can share the unique code of the game to friends to let them join.

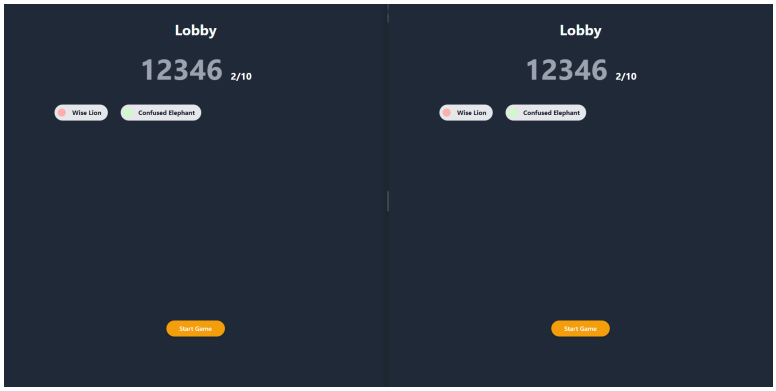

Figure 5: The lobby view for displaying players in the game before it starts, in the picture is two windows shown to show two joined players

If you want to join an existing game you type in the code you received from the host and simply press “Join”.

In the lobby you can see how many players are in the game. When everyone is ready anyone can start the game. You are then redirected to the game where a word will show up on the top. Below you can write your description of the word and either hit submit or wait for the timer to run out.

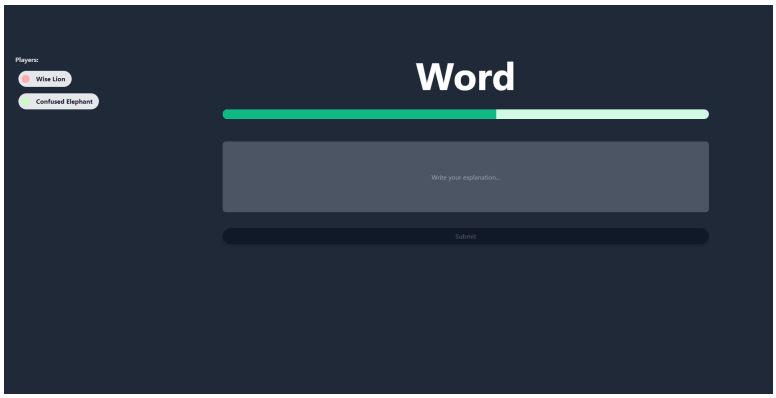

Figure 6: The view of the first stage of the game, presenting the word.

If everyone has submitted their answer or the timer runs out, you will be redirected to the page where you can see all the options to pick from. Here one of the answers will be the correct one.

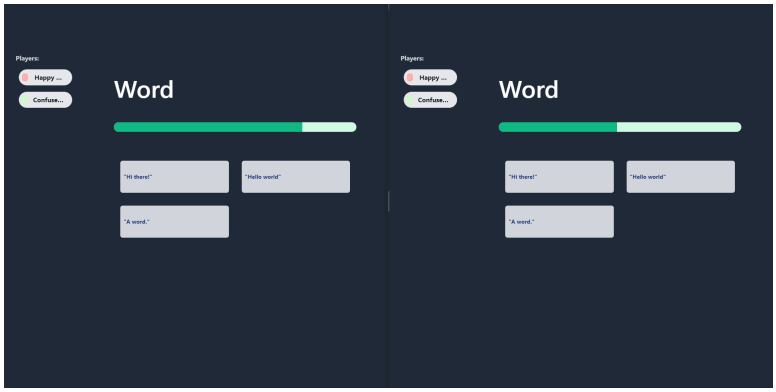

Figure 7: The view for selecting the answer you think is correct. In the picture is two windows shown.

You have to pick an answer before the timer runs out. After this, the correct answer is displayed and the points distributed.

After this, a scoreboard will show where you can see the leading player.

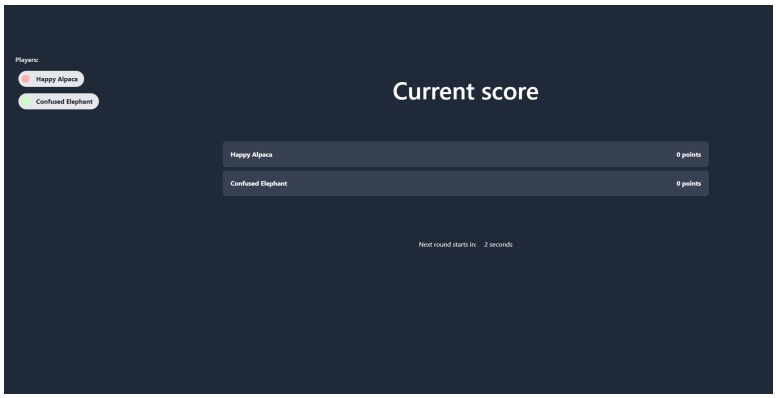

Figure 8: The view of displaying current score after one round.

4 Design

The backend has the following dependencies:

- Java
- SpringBoot
- PostgresSQL
- Lombok
- Jackson
- ReactJS
- Tailwind
- JUnit
- Deployed on Heroku

The project was divided into two separate sections, backend and frontend.

4.1 Frontend

The frontend separates four main views and then uses components to fill these views. These views are Game, Home, StartGame and Modify.

4.1.1 Connection with Websocket

The front end has a connection to the websocket endpoint for the current game identified by a pin code. The websocket is used for broadcasting the latest changes of a game to the front end. When a message is received its data is parsed into a JSON-object and then passed to the useGame hook to provide the application with global values. WebSocket messages are never sent from the frontend to the backend, which we will discuss later.

4.1.2 The useGame hook

To provide the application with up-to-date variables from the websocket on changes in the backend, we created a game context for the frontend. The whole application is wrapped with the GameProvider class and lets the application use the hook useGame.

```
const {
    globalGameState,
    pin,
    players,
    currentStateEndTime,
    currentWord,
    explanations,
```

```
    selectedExplanations,
    setGlobalGameState,
    setPlayers,
    setCurrentStateEndTime,
    setCurrentWord,
    setSelectedExplanations,
    setExplanations,
    setPin
  } = useGame();
```

The useGame hook let you access all variables and setters shown above wherever in the application and share their state. We use this solution to avoid passing around variables between components. This is a common use case in React `https://reactjs.org/docs/context.html`.

4.1.3 The Game View

The component Game represents the page for the url "/game/:pin" and keep all communication with the websocket for handling an active game. When recieving a message from the websocket its values are set to the corresponding variables in the hook useGame. The different states are defined and correspond to a particular component of the game view. The game view holds the components Lobby, Round and Summary which is rendered if the current state is LOBBY, PLAYING respectively END.

The Round component further holds four different components rendered depending on the current state of the game. These components will replace a part of the Round view to display whatever information is needed for the current state. An example of this would be the "PRESENT-WORD-INPUT-EXPLANATION" state which would replace the right side portion of the Round view to display the unusual word as well as an input field to write your explanation. With the state is also a end time for the state provided which is interpreted by the timer component to show the time left for an input. After the timer runs out, the websocket receive a state change to "SELECT-EXPLANATION" and displays the next view where you can select between the different answers. Using states means that we do not have to redirect to a new URL and render a new page every time there is a change, instead changing one portion of the screen.

The same way the components inside Round are rendered, the views for the Game view are rendered, switching between the LOBBY state, to ROUND state to lastly the END state.

4.1.4 Mutating and Creating Data

Ideally we would use the websocket for posting changes to the server. Unfortunately we encountered some problems with connecting the React frontend to the

websocket, used to modify session scoped data in the backend. After supervision with our supervisor we went for the solution to use post, update and delete requests to the server when mutating and creating data, and use the websocket exclusively for reading.

4.1.5 The Modify View

To easily perform simple CRUD-actions in the database we created the modify page. This page is currently not secured by any authentication due to time constraints and is found by clicking "edit" in the view for starting a new game. To secure this page is a requirement for a production build since its modifications will effect all users of the game.

4.2 Backend

The backend is divided into three main parts, the controller, the model and the websocket parts. We chose to implement the project with Spring. Springs layout is quite similar to Jakarta as it is also built on JavaEE but provides us with a simpler setup and deployment.

4.2.1 The Controllers

The controllers of the backend are constructed to respond to API calls and pass on the request to the model to get the desired actions. The controller is also responsible to throw correct error messages for bad requests or when values are undefined or null.

4.2.2 CRUD-actions from the Frontend

The backend of our project is an API only backend (with the exception of WebSockets). All communication to the backend is preceeded with "/api/v1". An example of an communication can be seen in Figure 9.

```
const joinGame = () => {
  const form = new FormData();
  form.append('name', getRandomName());
  axios
    .post(`/games/${pin}/join`, form)
    .then((res) => {
      console.log(res);
      localStorage.setItem('playerName', form.get('name'));
    })
    .catch((err) => {
      console.log(err);
    });
};
```

Figure 9: How the frontend communicates with the backend.

The call in the figure above gets interpreted by the figure below in the backend.

```
@PostMapping("/{pin}/join")
public ResponseEntity joinGame(
  @PathVariable("pin") String pin,
  @RequestParam("name") String name
) {
  var game :Game = getGame(pin);

  if (userBean.getPlayer() == null) {
    userBean.setPlayer(new Player(name));
  }

  if (!game.playerIsJoined(userBean.getPlayer())) {
    game.addPlayer(userBean.getPlayer());
  }

  game.notifyGameChangedObservers();

  return ResponseEntity.ok().build();
}

@PostMapping("/{pin}/start")
public ResponseEntity startGame(@PathVariable("pin") String pin) {
  var game :Game = getGame(pin);

  if (game.isStarted()) {
    throw new ResponseStatusException(HttpStatus.FORBIDDEN, "Game already started");
  }

  game.startGame();

  return ResponseEntity.ok().build();
}

private Game getGame(String pin) {
  var game :Game = gameManager.getGameByPin(pin);

  if (game == null) {
    throw new ResponseStatusException(HttpStatus.NOT_FOUND, "Could not find game");
  }
```

Figure 10: How backend interpret calls from frontend

4.2.3 The Model

The model package contains two sub-packages, the entity- and the game-package, and the Beans for managing scope of variables. The entity-package holds the inner construction of stored objects, defines whats the key and relations between each other. The game-package holds the logic for the game. The *GameManagerBean* and *UserBean* contains logic that is specifically scoped in a context.

4.2.4 Instantiating the Websocket

The package "websocket" in the repository contains the files to setup and broadcast messages with the websockets. The *WebSocketConfig* configures the websocket and creating the base endpoint. The class *GameChangedPushService* implements the interface *GameObserver* to observe the ongoing games and push messages to the endpoint with corresponding pin from the game with a change. The message sent is constructed by the class *GameResponse* which wraps the game and delivers the desired format for the frontend to interpret. The websockets are also secured with cross origin constraints which is setup in the *WebSocketConfig* class.

5 Application Flow

In this section we will go through the flow of the application in a few different scenarios.

Since we are using Spring's `SessionScope` attribute for identifiying users, we can not identify a session with WebSockets. Hence, we do communication that requires identification via regular Http requests. After researching, we found that it is possible to inject the HttpSession into a websocket request, but after talking to multiple supervisors, we ended up using ordinary Http requests for identification.

When the word "WebSocketWrapper" is references, it refers to the "combined" logic of the WebSocket and our `GameChangedPushService` class. The `GameChangedPushService` class is a `GameChangedListener` and notifies all clients in a specific game on a change to said game. It has the following method which sends a message to the clients:

```
1 @Override
2 public void notifyGameChanged(Game game) {
3   var pin = game.getPin();
4   simpMessagingTemplate.convertAndSend("/topic/messages/" + pin
        , new GameResponse(game));
5 }
```

So, the "WebSocketWrapper" is not actually a class, but it is used to simplify the protocol diagram.

5.1 Joining a Game

This section covers the flow for joining a game.. See Figure 11.

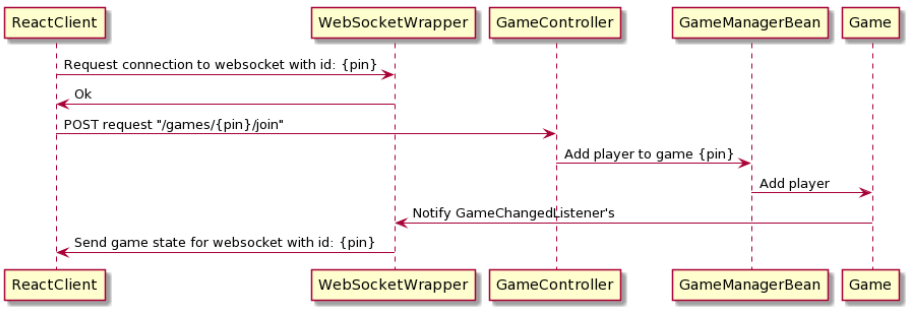

Figure 11: The flow of what happens when an user joins a game successfully.

5.2 Submitting an explanation

This section covers the flow when submitting your own explanation for the word. something. See Figure 11.

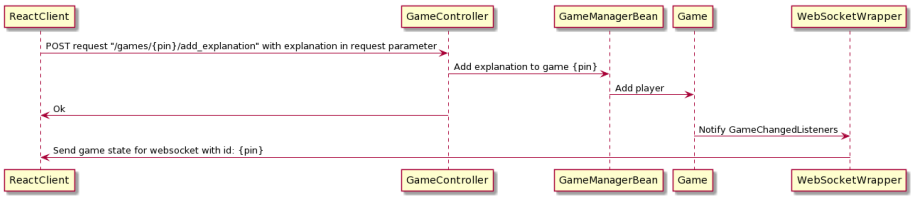

Figure 12: The flow of when an user submits an explanation successfully.

6 Package Diagram

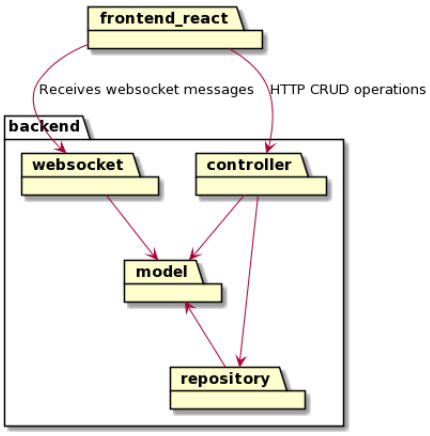

Figure 13: The package diagram for the application.

7 Code Coverage Report

Code coverage for the backend can be seen in Figure 14.

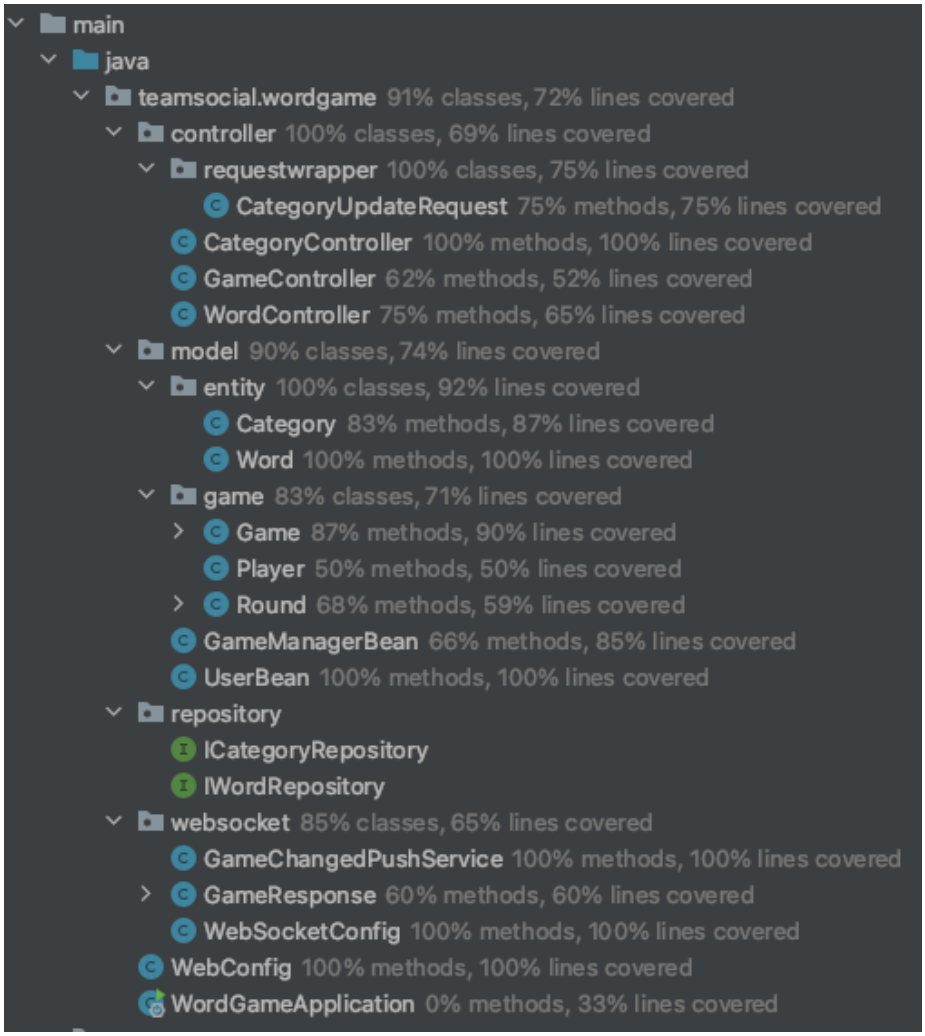

Figure 14: IntelliJ code coverage.

Due to time limitations, frontend testing was unprioritized. Although, there is a testing framework for the frontend that is working.

8 Model Diagram

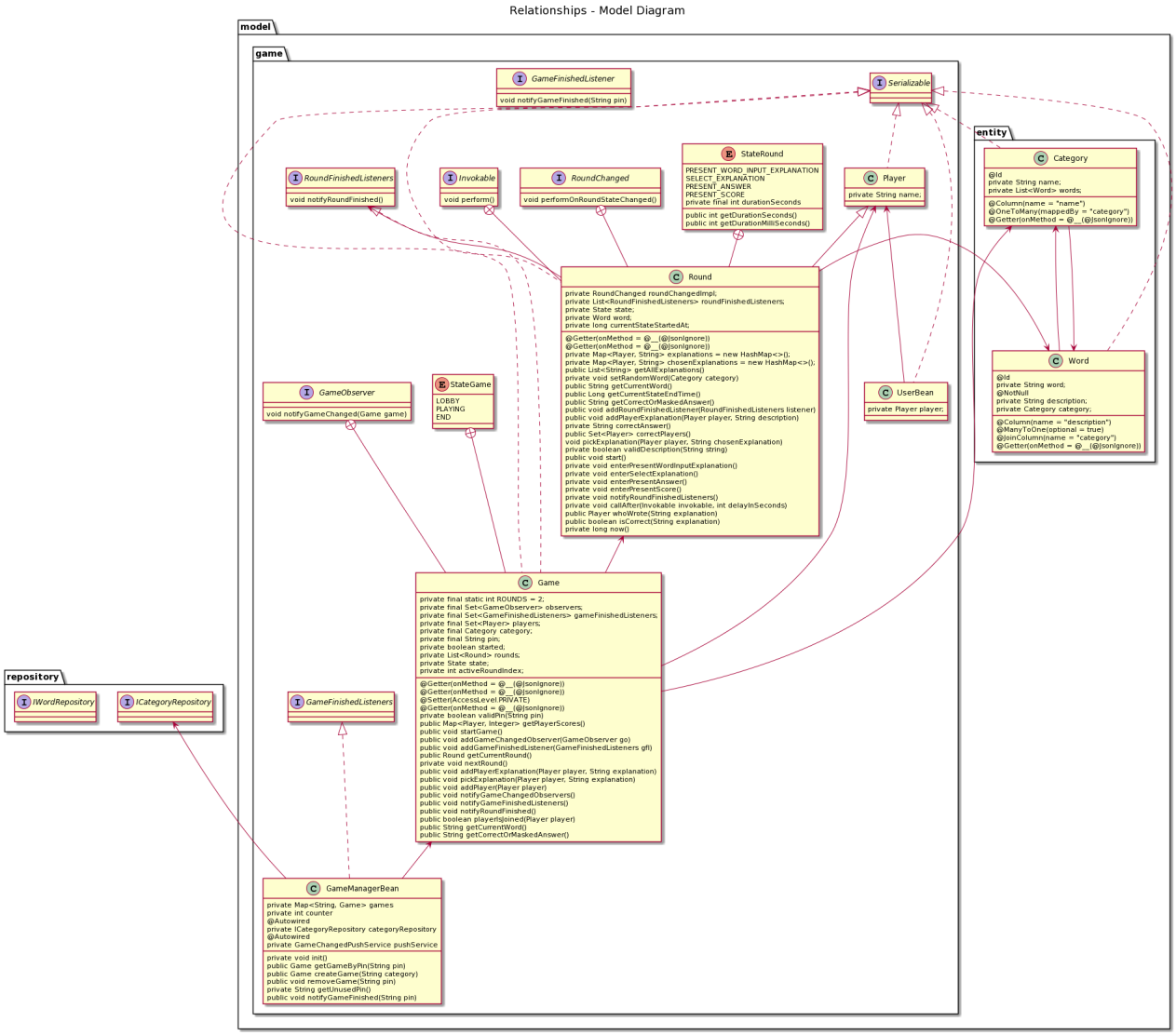

Figure 15: The model diagram for the application

9 Responsibilities

For this project we went back and forth between solutions for enabling real time updates between client and server. Also the members of the group had different goal of what to learn during the course. We discussed the best solution to take every ones wishes into account and the responsibilities got divided in a way where you got an extra responsibilities for the parts most interested in to learn. Early on when setting up the project, much work was done through pair programming with Live Sharing, but as the project progressed we worked more and more independently. Much work was done during Zoom sessions, where all members of the group discussed how to approach difficult tasks and gave input before making design decisions.

9.1 Theo Wiik aka backend wizard

Worked with most parts of the stack, and explored different approaches before the group made design decisions. Most of the work was put into writing the backend and the communication between the front and back-end. "Core" game model logic and tests.

9.2 Emanuel Enberg aka best practice instructorboi

Worked primarily with backend logic, in close collaboration with Theo. Early on in the project built a demo app, together with Theo, with web sockets in Elixir with Phoenix to show the potential of the framework for our supervisor, something which was scrapped for various reasons including the later requirement of Java as programming language.

9.3 Jonathan Heinin aka frontend dev guru

Worked primarily with the frontend. Having zero experience in the frameworks used but experience in similar frameworks, Jonathan worked on applying the design of the application. Jonathan worked with Jesper, integrating visual features to the application. Jonathan also managed and controlled the things required for the course such as the report and the requirements for the project.

9.4 Jesper Larsson aka the design god

Worked primarily with the frontend. With some prior experience with ui design and React, Jesper had a leading role in making the design and leading the work of the frontend. Jesper and Jonathan did a lot of pair programming together on these tasks with support from both Theo and Emanuel. Worked some on the backend for integrating it to the frontend and writing test classes. Jesper participated actively in group sessions over zoom with taking decisions and getting the project to proceed.