DAT076 Web Applications Project Report

Pontus Engström, Felix Erngård, Alexander Persson

March 12, 2024

1 Introduction

The application that we have created is called Chalmers Higher or Lower. It is a spin-off of the original higher or lower game, where the objective is to guess which one has the higher number of monthly searches according to Google 1. To make the game interesting, one alternative shows its amount of monthly searches and the other is hidden. This gives the user a reference point and a more enjoyable experience. If a user guesses correctly, they are awarded a point and if the user is incorrect, the game ends.

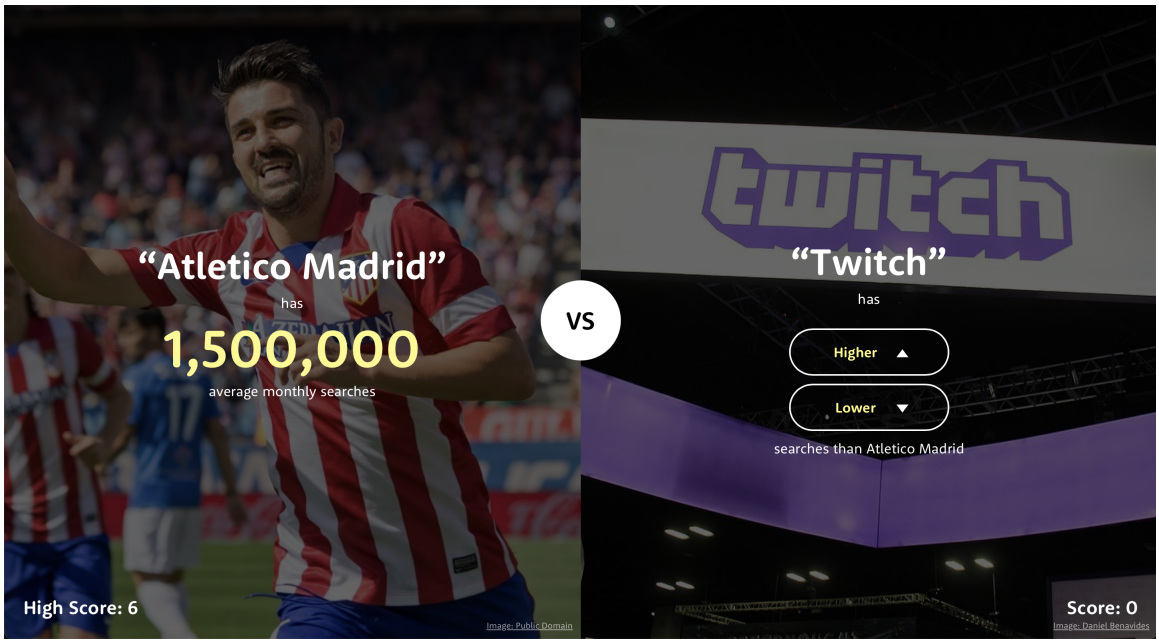

Figure 1: Picture of the original higher or lower game

1.1 Our application

Our application takes a similar approach as the original higher or lower game but with some twists. Instead of guessing which one has the higher number of monthly searches we decided to center it around Chalmers as we believed Chalmers students might be our target audience. Therefore, a user is to guess which Chalmers course has the higher rate of failure. Like the original game, whenever a user is correct their score updates and when they are incorrect the game ends.

In each game, the application tracks the player's score and saves it in a leaderboard, allowing users to see the progress of others and make the game more competitive.

Additionally, our application allows for a multiplayer experience, unlike the original higher or lower game. This was done to make the game more engaging as users can host their lobby, allowing for others to join.

1.2 Link to GitHub repo

https://github.com/Pontanami/DAT076-WebApp

2 Use Cases

- 1: The user can create an account to use in the application.
- 2: The user can click the trophy icon and see a leaderboard with a list of all players and their highscore.
- 3: The user can start their own singleplayer game.
- 4: The user can increase their score by clicking on the correct course while in a game.
- 5: The user can lose by clicking the incorrect course while in a game.
- 6: The user can lose the game by not answering within 15 seconds.
- 7: The user can get added to the leaderboard by finishing a game.
- 8: The user can update their highscore on the leaderboard by scoring higher than their current highscore.
- 9: The user can host a multiplayer game, which generates a unique game PIN.
- 10: A user can join the multiplayer game by entering the game PIN.
- 11: The host can start the multiplayer game for all joined users.
- 12: The host can start the next round for all joined users after they all have answered.
- 13: The joined users can update the local leaderboard of the multiplayer game by answering correctly.
- 14: The host can end the multiplayer game for all joined users.

3 User Manual

To install the app you need to:

1. Clone the repository by opening up a terminal, navigating to a folder of your choice and write git clone https://github.com/Pontanami/DAT076-WebApp.git
2. It is necessary to create a file ”dbPassword.txt” in the directory /server/src/db with the password given in canvas.
3. Open up two terminal windows. In the first navigate to the server directory of the application and write ”npm i” or ”npm install”. This installs all the necessary dependencies for the server. In the other terminal window navigate to the client directory and do the same. This step only needs to be done when starting the application for the first time.
4. Lastly start the app by writing ”npm run dev” in the terminal window with the server directory and write ”npm start” in the terminal window with the client directory.

   The app should be opened in the browser after a few seconds.

Using the app:

The first thing that shows up is a login page. An account is needed to use the app. For this a username and password is required, after which the user clicks ”Create Player” to create an account. After logging in the user arrives at the mainpage.

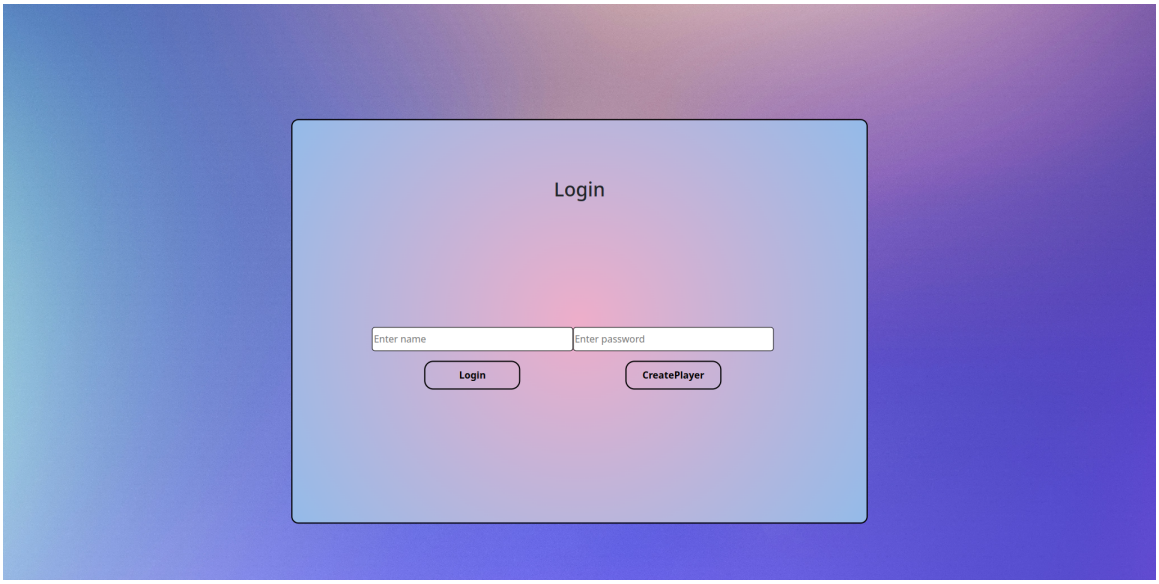

Figure 2: The login page.

Figure 3: The app's mainpage.

Here there are four buttons. The three at the bottom are "Host game" which lets you host a public game, "Join game" which lets you join a game that another user has created using the room's game PIN and "Singleplayer" which lets you play the game for yourself. The last button is in the top-right corner and lets the user see the leaderboard.

When playing the game the screen shows two different courses from Chalmers that people have completed. The failrate of the course to the left is hown and the one to the right is hidden. The user needs to click one of these within the time limit or else the game will be over and the user loses. If the course clicked has the higher failrate out of the two, the user gets a point and is presented with a new course to the right and the previous course to the right is moved to the left. This continues until the user loses. The first time a user loses its name and score is uploaded to the leaderboard, since it is that user's highscore. Then, if a user scores a new highscore the leaderboard is updated.

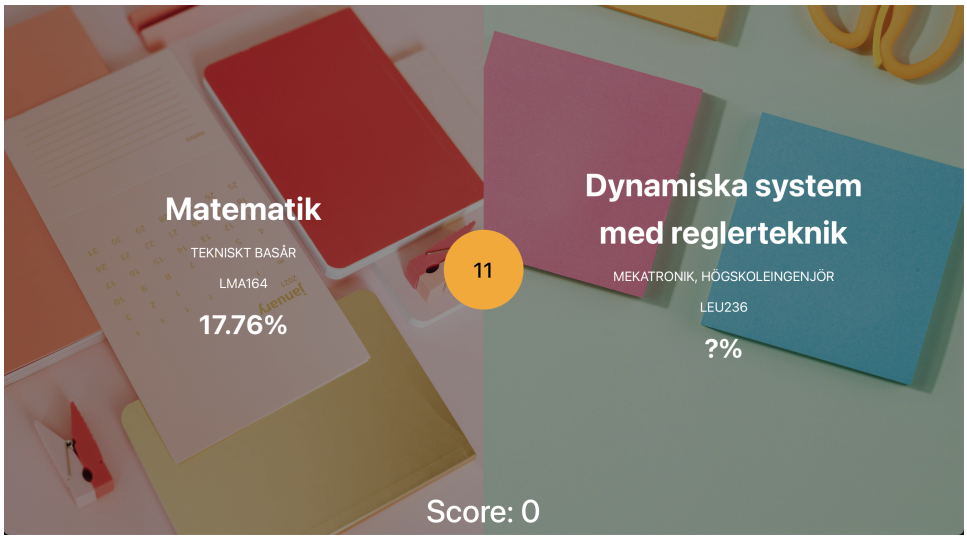

Figure 4: Shows how the app looks when a user is playing the game.

When playing multiplayer one user has to host a game while the others need to write the game PIN shown for the host in the input field when clicking ”Join game”. The host can start the game at any time and every player that has joined will start playing the game as normal while the host can choose to press next question and end game. During the game the hosts sees a leaderboard of the players playing in the hosted room.

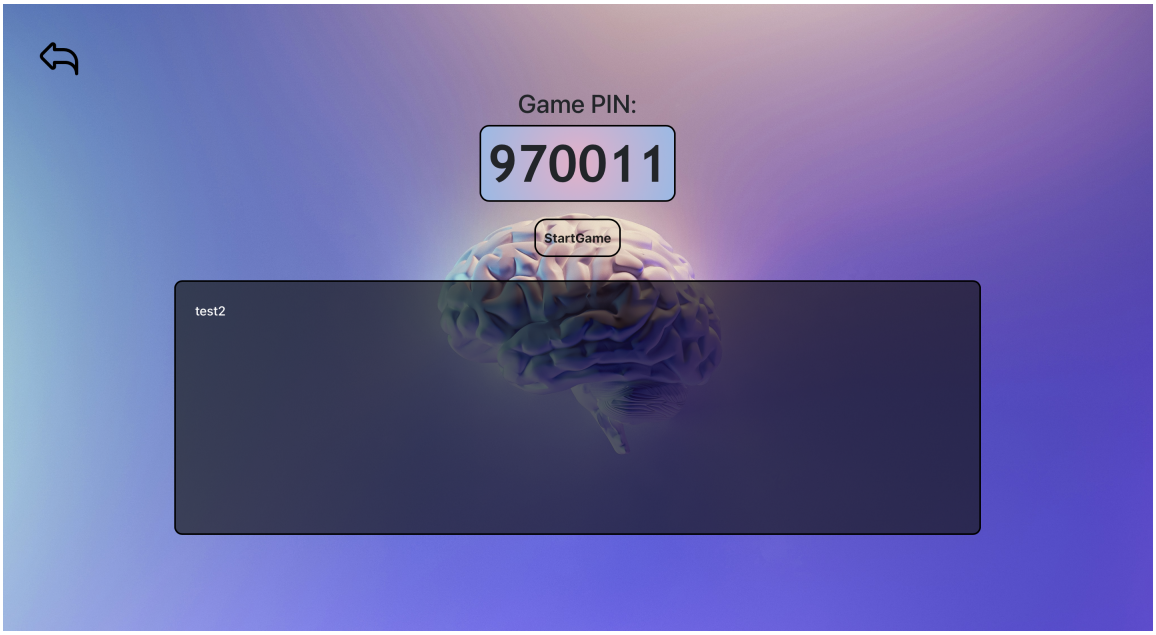

Figure 5: Shows the hosts point of view when a user has joined the room.

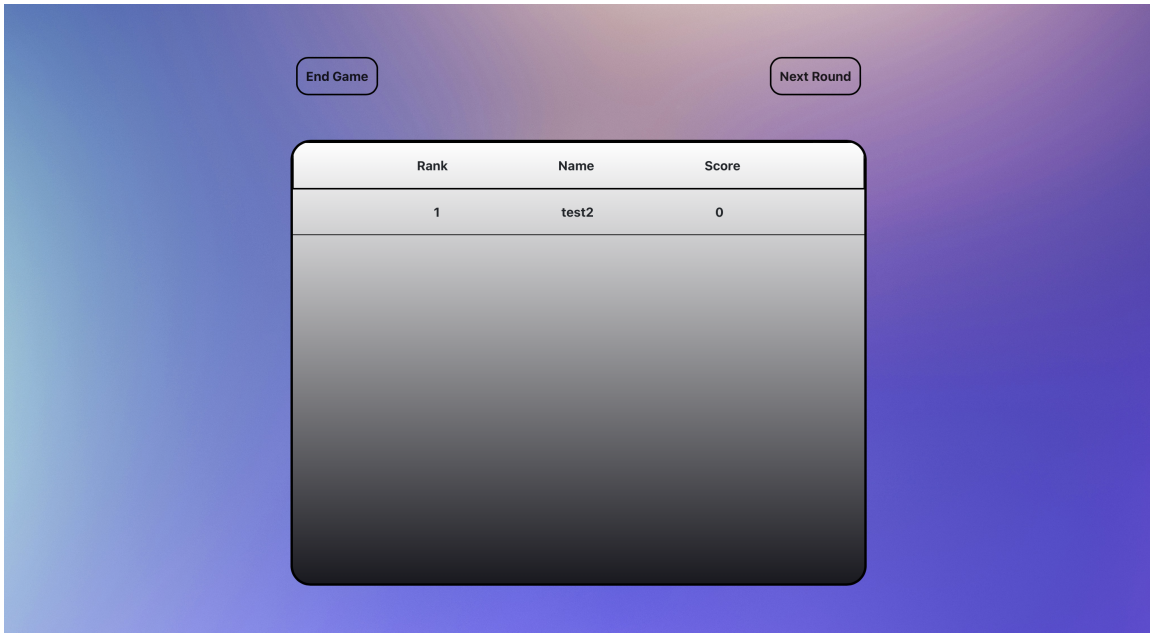

Figure 6: Shows the hosts point of view when the game has started.

4 Design

4.1 Libraries and Frameworks

Listed below are all the frameworks and libraries we used to build our web application.

- React with typescript (tsx) for frontend.
- A backend with service and router layer written in typescript.
- Server with node express.
- MongoDB database.
- Testing with jest.
- MongoDB memory server for database testing.
- SocketIO for web sockets which enables multiplayer games and real time updates for all players in a multiplayer game.
- Axios to handle API calls.
- React bootstrap for some application components.
- User event testing library for simulation of user interactions in testing.

4.2 System architecture

This section will describe the system as a whole and which responsibilities different parts of the system has.

4.2.1 Client

- Error folder
  - Errorhandling - function responsible for handling errors, converting them to displayable messages. Takes the error as an argument.
  - ErrorScreen - Class responsible for displaying the error screen and call the app component to render the error screen whenever an error has arose. Takes a string as an argument representing the error message and a function to update what the app component is displaying.
- Home folder
  - Home - Class responsible for displaying the home screen and its functionallity. It redirects users whenever they want to host, join or play a singleplayer game and calls the backend, telling it to create a new player. Takes in an errorHandler to be called whenever error arise.
  - Login - Class responsible for the login screen. This includes displaying error messages if something went wrong, and sending API calls to the backend if a user wants to create an account or log in.
- Leaderboard folder
  - DisplayLeaderboard - Reusable class that displays both the overall leaderboard and the local leaderboard of a multiplayer game. Takes a list of players as arguments and calls the leaderboardplayer component to create new players in the leaderboard.

  - Leaderboard - Class responsible for diplaying the whole of the leaderboard. Fetches players when created and calls DisplayLeaderboard to display the leaderboard properly. Takes errorHandler as an argument, used if something goes wrong
  - LeaderboardPlayer - Class responsible for displaying a single player on the leaderboard.
- Play folder
  - DisplayCourse - Class responsible for displaying the two clickable courses. Handles when a user clicks on a course and sends an API call to the backend posting the user's answer. Takes the courses we want to display, and several functions called whenever the user has answered either correctly or incorrectly.
  - GameOver - Class responsible for displaying the gameover screen
  - Multiplayer - Class responsible for handling multiplayer-specific tasks, for example how to fetch courses, which differs from how you do it in singleplayer. Takes errorHandler as an parameter.
  - Playscreen - Class responsible for handling common play-specific tasks, such as updating a users score. Also creates the core of the playscreen and calls DisplayCourse to display the specific courses. Takes methods a list of courses and methods for handling correct and incorrect guesses as arguments.
  - Singleplayer - Class responsible for handling the singleplayer-specific tasks. This includes updating questions whenever a user guesses correctly. Takes errorHandler as an argument.
- App - Class responsible for creating routes to different parts of the project. Displays error messages if any arise during a users session.
- BackButton - Reusable class which provides a way of returning to the homescreen. Used in for example the leaderboard, join game and host game.
- CurrentUser - Responsible for holding information of the current user, its id and username. Used throught the application to fetch information from the logged in user.
- Host - Class responsible for providing services needed to host a multiplayer game. This includes create a gamepin, update questions and display a local leaderboard updated in real-time. Takes errorHandler as an argument.
- HostPort - Facilitates switching between ports, as an update here changes all the ports in which the requests are sent to.
- ICourse - interface for defining the characteristics of a course. Used when creating courses after a response from the server.
- index - Creates the application.
- IPlayer - interface for defining the characteristics of a player throughout the application.
- Join - Class responsible for allowing users to join others games. Displays fields where a user can input gamePins of rooms they want to join and sends requests whenever a user wants to join a room. Takes errorHandler as an argument
- JoinScreen - Class responsible for displaying the screen a user sees when they have joined and are waiting for the user to start the game.
- Socket - Responsible for creating a socketconnection to the backend on a specified port

4.2.2 Server

- Db - This folder contains MongoDB specific database implementation such as setting up the connection to the database and creating the different schemas for the database objects. This is also where the uri and password for the database needs to go. The ”mocks” folder is used to create a mock database during testing using MongoMemoryServer.
- Router - Each class in router corresponds to a specific service which handles any specific functionality that needs to be called. The specifics of the router layer is mentioned later in the ”API specification” section.
- Service - The folder where the backend functionality of the app is located. Every implementation class here has an interface variant which is used by the router to follow the Dependency Inversion principle. Every service also has a test file with the same name.
  - course - A singleton class responsible for handling course related tasks such as creating a course object from the model, retrieving those created and comparing the failrate of two courses and returning a boolean for whether or not the one that was ”selected” was higher.
  - game - A class that handles the functionality of a game with methods such as creating a game, getting the questions of the game, start the next round, etc.
  - leaderboard - The class managing the adding, updating and retrieving from the leaderboard. As it is right now, this implementation (using the ILeaderboardService interface) is using MongoDB for its methods.
  - multiplayer - The class dealing with the multiplayer aspect such as joining a multiplayer room and retrieving all the players currently in a room. Has a game composite that handles the game aspect.
  - player - A singleton managing the creation and modification of the player object which is used inside the game. Creating a player, retrieving a player and updating the score is done here.
  - singleplayer - The class dealing with the singleplayer. Does not do much else besides using the game class through composition and connecting a player to it.
  - user - The class managing the registration of users and the login method. This current implementation is using MongoDB.

4.3 API specification

4.3.1 Course

Post to ”course/answer”

- Arguments
  - codeClicked: Argument of type string representing the course code of a player's clicked course.
  - otherCode: Argument of type string representing the other course code, not clicked by a player.
  - playerId: Argument of type number representing the id of the player clicking a course.
- Response

- Success 200: If the API call is successful without errors a 200 response is sent back along with a boolean representing if the user's clicked course is the one with a higher rate of failure.
- Error 400: A statuscode 400 is returned if one of the arguments is either undefined or of the wrong type. A string is returned as a response, telling the user what went wrong.
- Error 500: If something went wrong in the service layer or the connection to the server is refused a 500 response is returned along with a string describing the error.

4.3.2 Game

Post to "game/update"

- Arguments
  - gameId: Argument of type number representing the id of the game we want to modify.
- Response
  - Success 201: router responds with a 201 request if the update was successful. If so it sends back the two next courses.
  - Error 400: Statuscode 400 is returned if a argument is of the wrong type. A string is returned as a response, telling the user what went wrong.
  - Error 500: Responds with a 500 error if something went wrong while processing the request in the service-layer or if no response was provided from the server. Returns a string representing the error message.

Get to "game/"

- Arguments
  - Parameter id: Argument of type number representing the id of the game we want to fetch the current questions from.
- Response
  - Success 200: Router responds with a 200 request if the request was successful, returning the current questions of the game.
  - Error 400: Statuscode 400 is returned if a argument is undefined or gameId is negative. A string is returned as a response, telling the user what went wrong.
  - Error 500: Responds with a 500 error if something went wrong while processing the request in the service-layer or if no response was provided from the server. Returns a string representing the error message.

4.3.3 Leaderboard

Get to "leaderboard/players"

- Takes no arguments
- Response
  - Success 200: Router responds with a 200 request if the request was successful, returning all players on the leaderboard in descending order according to their score.

- Error 500: Responds with a 500 error if something went wrong while processing the request in the service-layer or if no response was provided from the server. Returns a string representing the error message.

Post to "leaderboard/"

- Arguments
  - id: Argument of type number representing the id of the player we want to either update or add to the leaderboard.
- Response
  - Success 201: router responds with a 201 request if the update was successful. If so, it sends back an array consisting of the updated version of players on the leaderboard.
  - Error 400: Statuscode 400 is returned if a argument is of the wrong type. A string is returned as a response, telling the user what went wrong.
  - Error 500: Responds with a 500 error if something went wrong while processing the request in the service-layer or if no response was provided from the server. Returns a string representing the error message.

4.3.4 Multiplayer

Post to "multiplayer/"

- Arguments
  - hostId: Argument of type number representing the id of the game's host.
- Response
  - Success 201: router responds with a 201 request if the game was created successfully. If so, it sends back the id of the newly created multiplayer game.
  - Error 400: Statuscode 400 is returned if a argument is of the wrong type. A string is returned as a response, telling the user what went wrong.
  - Error 500: Responds with a 500 error if something went wrong while processing the request in the service-layer or if no response was provided from the server. Returns a string representing the error message.

Post to "multiplayer/addPlayer"

- Arguments
  - gameId: Argument of type number representing the id of the game we want to add the player to.
  - playerId: Argument of type number representing the id of the player we want to add to the game.
- Response
  - Success 201: router responds with a 201 request if the player was successfully added to the multiplayer game. If so, it sends back a boolean telling the user of its success.
  - Error 400: Statuscode 400 is returned if a argument is of the wrong type. A string is returned as a response, telling the user what went wrong.

- Error 500: Responds with a 500 error if something went wrong while processing the request in the service-layer or if no response was provided from the server. Returns a string representing the error message.

Get to "multiplayer/"

- Arguments
  - Parameter id: Argument of type number representing the id of the game we want to fetch the players from.
- Response
  - Success 200: Router responds with a 200 request if the request was successful, returning the players of the game sorted by score in descending order.
  - Error 400: Statuscode 400 is returned if a argument is undefined or gameId is negative. A string is returned as a response, telling the user what went wrong.
  - Error 500: Responds with a 500 error if something went wrong while processing the request in the service-layer or if no response was provided from the server. Returns a string representing the error message.

4.3.5 Player

Get to 'player/'

- Arguments
  - Parameter id: Argument of type number representing the id of the player we want to fetch.
- Response
  - Success 200: Router responds with a 200 request if the request was successful. Returns the player we fetched.
  - Error 400: Statuscode 400 is returned if a argument is undefined or id is negative. A string is returned as a response, telling the user what went wrong.
  - Error 404: Is returned of we could not find the player.
  - Error 500: Responds with a 500 error if something went wrong while processing the request in the service-layer or if no response was provided from the server. Returns a string representing the error message.

Post to 'player/'

- Arguments
  - id: Argument of type number representing the id of the user in which we want to tie the player to.
  - name: Argument of type string representing the name of the player we want to create.
- Response
  - Success 201: router responds with a 201 request if the player was successfully created. If so, it sends back the player object.
  - Error 400: Statuscode 400 is returned if a argument is of the wrong type. A string is returned as a response, telling the user what went wrong.

- Error 500: Responds with a 500 error if something went wrong while processing the request in the service-layer or if no response was provided from the server. Returns a string representing the error message.

4.3.6 SinglePlayer

Post to "singleplayer/"

- Arguments
  - playerId: Argument of type number representing the id of the player creating the singleplayer game.
- Response
  - Success 201: router responds with a 201 request if the singleplayer game was successfully created. If so, it sends back the id of the singleplayer game.
  - Error 400: Statuscode 400 is returned if a argument is of the wrong type. A string is returned as a response, telling the user what went wrong.
  - Error 500: Responds with a 500 error if something went wrong while processing the request in the service-layer or if no response was provided from the server. Returns a string representing the error message.

4.3.7 User

Post to "user/signup"

- Arguments
  - username: Argument of type string representing the name the user wants to be displayed
  - password: Argument of type string representing the user's password, used for logging in
- Response
  - Success 201: router responds with a 201 request if the player was successfully created. If so, it sends back a tupple with the id of a user and their name.
  - Error 400: Statuscode 400 is returned if a argument is of the wrong type. A string is returned as a response, telling the user what went wrong.
  - Error 500: Responds with a 500 error if something went wrong while processing the request in the service-layer or if no response was provided from the server. Returns a string representing the error message.

Post to "user/login"

- Arguments
  - username: Argument of type string representing the name the user we want to login with.
  - password: Argument of type string representing the user's password.
- Response
  - Success 200: router responds with a 201 request if the player was successfully logged in. If so, it sends back a tupple with the id of a user and their name.

- Error 400: Statuscode 400 is returned if a argument is of the wrong type. A string is returned as a response, telling the user what went wrong.
- Error 404: If the user wasn't found a 404 statuscode along with a message is sent.
- Error 500: Responds with a 500 error if something went wrong while processing the request in the service-layer or if no response was provided from the server. Returns a string representing the error message.

5 Responsibilities

Listed below are each of the responsibilities we've had during the project. Almost all time spent on the project was sitting together so we all helped each other with many of the tasks during the project.

Alexander

- GUI design, including:
  - Building the host page
  - Building the join page
  - Building the wait for host screen
  - Designing the login page
  - Designing the game over screen
  - Designing the multiplayer game ended by host screen
  - Designing the clickable courses that a user clicks while playing
  - Fixes on the leaderboard, homepage, playscreen and host next round/end game screen
- Adding the multipage functionality, i.e. enabling navigation between pages in the application.
- Multiplayer router
- Player and multiplayer model
- Fixes in multiplayer service
- Generating and displaying game PIN for multiplayer games
- Creating and pushing the initial react project. Therefore, my git contribution is a bit bloated where it says I added over 20,000 lines of code. However, most of that is from just creating the project

Felix

- Implemented multiplayer functionality with socket.io:
  - Able to host game
  - Join game
  - Change question as host
  - End game for all players
  - See players who has joined in real-time
  - Start multiplayer game
- Implemented user-features, such as login and create user both in the frontend and backend
- Created the logic for playing the game in react
- Created tests for router, frontend tests for leaderboardplayer, host screen and join screen
- Implemented, together with Pontus, game, singleplayer, player and leaderboard in the service layer
- Created backend for multiplayer

- Created or re-engineered most of the routers classes.
- Helped design the frontend, mainly homescreen
- Created and implemented the design of the playscreen
- Created error handling for the application

Pontus

- Implemented, together with Felix, game, singleplayer, player and leaderboard in the service layer
- Implemented course in the service layer.
- Created tests for service and for the play folder in the frontend
- Helped creating alot in the router layer, focusing on course, player, game and some tests
- Created the leaderboard design
- Implemented the leaderboard for the host using web sockets
- Helped make the screen responsive during the game.
- Implemented timer and score counter in the game