DAT076 Web applications Report Group 14

Ali Aziz, Oscar Broborn, Mhd Suheib Shahin, Jesper Vesanen

March 18, 2025

Contents

In this section of the report, a brief description of the application will be presented. Additionally, a list of use cases for the application will also be brought forward. Here is the Github repo: `https://github.com/Willard-VanDine/WebsiteProject`

1 Introduction

The group set up a GitHub repository in order to maintain a stable version control, allowing the group to work in parallel over the course of the project. During the course, the group decided to create a web application that allowed users to play a simple game. The game that was decided was Rock, paper, and scissors. The app therefore allows a user to register and play best-of-five games of Rock, paper, and scissors, as well as track their scores. Rock, paper, and scissors was chosen due to the simple nature of the game, which allowed the group to focus on good design rather than complex algorithms.

For example, the group focused on creating a reactive UI that should be easy to navigate through for the users. An example of this is the component design, which only re-renders a specific part of the website whenever a user navigates throughout the application.

Another example of good design was the focus on the quality of the code rather than its complexity. Where high quality code is defined as code that is extensible, closed for modification, loosely coupled, and also has high cohesion. However, optimizing these four parameters can be quite the challenge as well, as it might reduce the simplicity of the code significantly. In order to address this issues, a balance between good practices and simple design was carefully maintained. An example of this is that the application has been designed so that it is possible to add more games other than Rock, papers and scissors, with only minimal changes in the backend.

The final focus has been to ensure security for the application itself and for its users. This is done by limiting user privileges and capabilities, thus reducing the risk of malicious actions, while also not tempering with accessibility for normal users.

2 Use Cases

In order to ensure that the application functions in accordance with the intended design, the following use cases are used. These use cases will later be demonstrated and compared to the written requirements during the project presentation:

1. A User should be able to subscribe to a game, such as ”Rock, Paper, Scissors” or even other games if such were to be added in the future
2. A User should be able to start and play a match of ”Rock, Paper, Scissors” in a best-of-five format
3. A User should be able to see the current score of an ongoing match in real time
4. A user should be able to see their amount of total game wins and losses along with other statistics such as win rate
5. A user should be able to leave or close the application during an unfinished match and come back to continue playing later without losing progress of said match

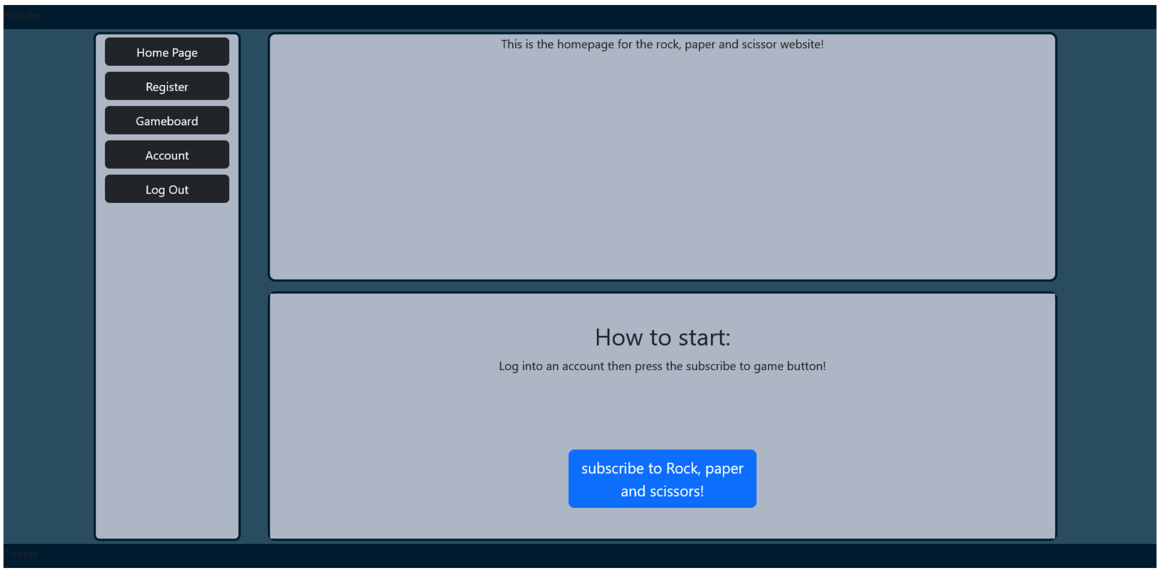
3 User manual

This following section will describe how to install, and then use the application from the point of view of a user.

3.1 Installing the app

- Step 0: Install Git, Node.js® and Docker Desktop on your system
- Step 1: Open a terminal in a directory of your choice
- Step 2: Clone the repository to a folder using **git clone https://github.com/Willard-VanDine/WebsiteProject.git**
- Step 3: Open the *server* and *client* directory in two separate command prompts
- Step 4: In the *server* directory, create a file with the name *.env* and edit it to contain the following text on the first line: **SESSION_SECRET= "*text of your choice, within quotation marks*"**
- Step 5: In each command prompt install all the necessary dependencies using the command **npm install**
- Step 6: Launch *Docker Desktop* and start a local Postgres database on port 5432.
- Step 7: After installing the dependencies run **npm run dev** in each command prompt
- Step 8: Enter the website using the link provided in the *client* terminal.

3.2 Using and navigating the app



This is the homepage for the rock, paper and scissor website!

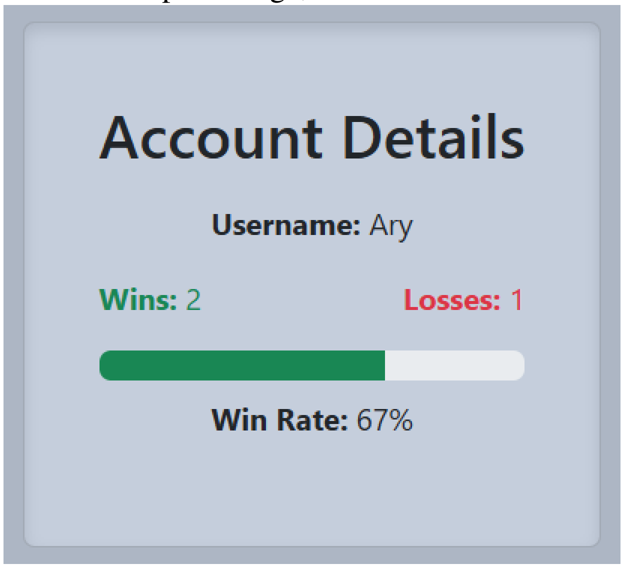
How to start:

Log into an account then press the subscribe to game button!

This is the homepage of the application; it explains how to operate the website properly, giving the user guidelines on how to navigate the website. On this page, the user can find the ”subscribe to game” button. If the user is not logged in, it will merely give you an alert that tells you to log into an existing account, or to register a new account. However, if the user is logged in, and if the user clicks on the button, it will subscribe them to the rock, paper, and scissors game, with a subscription being a prerequisite for a user to actually play the game. If the user is already subscribed to the game, they will get an alert that tells them that they are already subscribed.

On the sidebar on the left, whenever the user is not logged in, routes to register and login are available. Whenever the user clicks on these routes, the huge container on the right will be switched out and render another component. If the user clicks on the register route, it will load a component that allows the user to register an account. There are two input fields that represent username and password, respectively. Whenever the user has filled in the two input fields, they can click on the register button to actually register an account. However, if the username has less than three characters or more than twenty characters, an alert will pop up telling the user to register with proper inputs. There is also red text underneath the input fields whenever the requirements are not met. The login page is similar; it has two input fields and also a button that is supposed to be clicked whenever the user wants to log into an existing account. If the user puts in the wrong input, they will be informed of such a mistake through a pop-up.

Whenever the user is logged in to an account, the sidebar updates, giving access to new pages while also removing access to the login page. The two new routes that are added to the sidebar are the gameboard route and also the account route. The account page simply displays the logged in account, as well as their total wins and losses, alongside a winrate in percentage, as shown below:



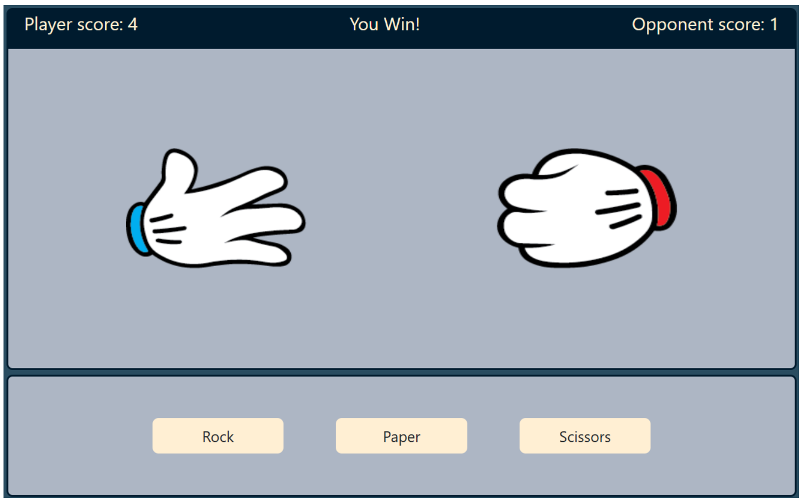
Account Details

Username: Ary

Wins: 2 **Losses:** 1

Win Rate: 67%

Whenever the user clicks on the gameboard route, it will load the gameboard component, which includes a gameboard area that tells the current score of both the player and the opponent; both of these start at 0. There is also a button container at the bottom which includes three buttons representing ”Rock”, ”Paper” and ”Scissors” respectively. If the user is subscribed to the game and clicks on one of these buttons, then an image will appear on the gameboard area, representing the choice that the user made. So if the user chose ”Rock” then an image of a hand forming a rock will appear. After the user has clicked one of the buttons and made a choice, then the computer will make a random choice as well, which will be visualized on the gameboard similarly to how the user's choice is visualized. Worth noting is that if the image can not be loaded, it will instead write out the word of the choice instead of the image. Then just as the rules in rock, paper, and scissors, a winner of the round is determined depending on the choice of the player and the choice of the computer. Then afterwards, the scores are updated respectively, depending on who wins the round. The winner of the round will then be printed onto the gameboard, such as if the user wins, it will write 'player wins', if it is a draw it will print out 'Draw', and 'Opponent wins' if the user loses. Finally, if one of the player or the opponent reaches 5 points, they will win a game, which will give the user an accountWin or an accountLoss depending on the result which is later used in the account page. Worth noting that all of these features only work if the user is logged in, if they are not the page does not render and instead it will redirect the user to the homepage. Also if the user switches site and re-enters the gameboard, the correct round score will update. The picture below shows the gameboard:



The final page is from the account router. This page is simple, it automatically loads the account stats of the user and displays them on the page. This includes the username and the amount of wins the user has and the amount of losses the user has. It also shows the win rate of the user. This page does not render if the user is not logged in, and instead it navigates the user to the homepage. This is a safety measure to make sure a user can not try to access the site manually through entering the url in the browser.

4 Design

This section will provide a technical description of the app's construction. The following will be included: the libraries that are being used, frameworks that are integrated, and any other relevant technologies.

4.1 Components

The apps consist of numerous components which begin with the main component *main.tsx* whose purpose is to establish the layout of the application, which consists of a header, content, and footer.

4.1.1 Header

The header, as the name suggests, is currently on top of the page; however, as of now, its function is to add additional depth to the app's visual appeal.

4.1.2 Content

Next part is the *Content.tsx* component which comprises of *Sidenav.tsx* and *Container.tsx* The content objective is to render the application's inner components. It sets up the necessary content area and routes for the application to navigate between and load *Container.tsx*. The state that is being used is *isLoggedIn* of type boolean and its corresponding function to updates its state. The component's state relays on *useAuth()* hook for login status for its initial state. The state *isLoggedIn* is being passed down as a prop to the following routes' /gameboard, /login" and /account, the exception being /login which also requires *setIsLoggedIn*. Additionally, Sidenav requires *isLoggedIn* as its prop.

4.1.3 SideNav

Next in order is *Sidenav.tsx* which holds multiple Navlinks to the different routes for easier navigation throughout the various components. It utilized the prop passed down by the content to determine the user's current status and displaying its corresponding navlinks and button. This is achieved through the function call *logOut* that sends a GET request to the backend to log out the user, and when the client receives status code 204 then the user is successfully logged out. The current implementation of Sidenav does not contain any states, nor props.

4.1.4 HomePage

Homepage.tsx is constructed as the main page of the application, as of now, it has two purposes; one is to display any relevant information to the use, and the second is to subscribe the user to the game. The subscription feature is intended to increase the application abstraction, thus increasing its extendibility without requiring changes to the main code when additional games are added down the line. The subscription action is performed through a POST request to the backend server. The client is successfully registered to game when a 201 status code is received. However, if the processes were to return 500 status code it would indicate an unsuccessful attempt to register for a game, an alert will be displayed corresponding to each status code. Homepage does not contain any states, or props.

4.1.5 Gameboard

This component is the central part of the application. This is where the game is being displayed and interacted with. it consists of two parts: the upper section where the game is being rendered, while the lower section contains the button which the user could interact with. The result will be a part of the upper section, where the result of the ongoing round will be displayed. The current implemented game Rock paper, scissors will have a corresponding picture for each shape to add further visual depth. The result of the round will be presented as an alert on top of the page to indicate whether the user lost or won. Gameboard uses props sent by *Content* components to evaluate whether the current status of the user is sufficient to render the page. The component send two https request, one is GET request to retrieve game score and a POST-request to the backend server to make a move, in both cases 2** status code indicate a successful request. However, a status code of 4** expresses an unsuccessful request caused by an incorrect input by the user. Gameboard does not use any state in the current implementation and does not have any children components so it does not have any props to send.

4.1.6 Login

The login page handles the processes of user logins. It consists of two inputs fields: first for the username and the second for the password. Towards the bottom of the page, there is a button that call on the function *handleLogin*, it sends a POST-request to the backend server. A result of a 200 status code indicates a successful login, while a 401 status code expresses the opposite. In order to maintain the state of the username and the password, components have two states *username* and *password*, and their *setUsername* and *setPassword*. It does not however have children and therefor no props.

4.1.7 register

It consists of two inputs fields: first for the username and the second for the password. Towards the bottom of the two, there is a button that call on the function *handleLogin*, it sends a POST-request to the backend server. a Result of a 200 status code indicates a successful login, while a 401 status code expresses the opposite. In order to maintain the state of the username and the password, component have two states *username* and *password*, and their *setUsername* and *setPassword*. It does not however have children and therefor no props. Register component does not have any props.

4.1.8 Account page

Account page purpose is to display the result of all the games ever play, presented in a visually pleasing way. The page shows the win rate, total lose, total win, as well as, the username, all this data is contained within a card container. In order to retrieve these values, a function is called *getAccount* that fetch these values from the database. These values are being stores in state of *account* and its corresponding function *setAccount* to manipulate its value. The page uses props from the *Content* in order to determine if the current status of the user is appropriate for the page to render, otherwise the user is redirected. Account component does not have any children, this does not require sending any props.

4.1.9 Footer

The footer, as the name suggests, is currently at the bottom of the page. Its current state is similar to the header, as it is there for display purposes.

4.2 Backend API

This section presents the specification of the backend API, providing descriptions and details of the requests that are accepted for each API request, alongside the responses for valid and invalid requests.

4.2.1 Account Router

GET /getAccount This request retrieves the account details of a user's Account, including username, account wins, and account losses.

- Verb: GET
- Endpoint: /getAccount

- Body: Request body is derived from the session object, which contains the username of the user to get an account from.
- Responses: **200 OK**, returning username, accountWins, and accountLosses. **401 Unauthorized** if no user exists or user not logged in. **500 Internal Server Error** if there is a server issue.

GET /check-session Checks whether a user is logged in or not by verifying a session cookie

- Verb: GET
- Endpoint: /check-session
- Body: Request body is derived from the session object, which contains the username of the user to validate whether it is logged in from.
- Responses: **200 OK**, returning {"loggedIn":true} if logged in, otherwise false with **401 Unauthorized**. Returns **500 Internal Server Error** if server issue is encountered.

GET /logOut Logs the user out by clearing the session cookie.

- Verb: GET
- Endpoint: /logOut
- Body: Request body is derived from the session object, which contains the username of the user.
- Responses: **204 No Content** if the user is logged out successfully, as we don't expect any data. **401 Unauthorized** if the user is not actually logged in. **500 Internal Server Error** if there is an issue with the server.

POST / Registers a new user with the given username and password

- Verb: POST
- Endpoint: /
- Body: {"username":"name", "password": "pass"}
- Responses: **201 Created** if the account is created successfully. **400 Bad Request** if username is less than 3 characters or greater than 20 characters, or password is less than 5 characters or greater than 50 characters. **401 Unauthorized** if account already exists. **500 Internal Server Error** if there's an issue with the server.

POST /login Logs in the user with the given username if the password is correct.

- Verb: POST
- Endpoint: /login
- Body: {”username”:”name”, ”password”:”pass”}
- Responses: **200 OK** if the login is successful. **400 Bad Request** if the user is already logged in. **401 Unauthorized** if the password(or username) do not both match an account. **500 Internal Server Error** if there are issues with the server.

4.2.2 Gamestate Router

GET / Gets the current game state of the logged in user

- Verb: GET
- Endpoint: /
- Body: Request body is derived from the session object, which contains the username of the user.
- Responses: **200 OK**, returning the gamestate {”playerScore”:num, ”opponentScore”:num, ”gameName”:”name”, ”gameThreshold”:num, ”winnerOfGame”:winnerOfGameEnumValue}. **401 Unauthorized** if the user is not logged in. **400 Bad Request** if the user is not subscribed to the game. **500 Internal Server Error** if there is an issue with the server.

PATCH / Make a move in the game for the logged-in user.

- Verb: PATCH
- Endpoint: /
- Body: {”playerChoice”:Choice}
- Responses: **201 Created**, returns the updated game state if the move is successful. **401 Unauthorized** if a user is not logged in or subscribed to the game. **400 Bad Request** if the player choice is invalid. **500 Internal Server Error** if there are server issues.

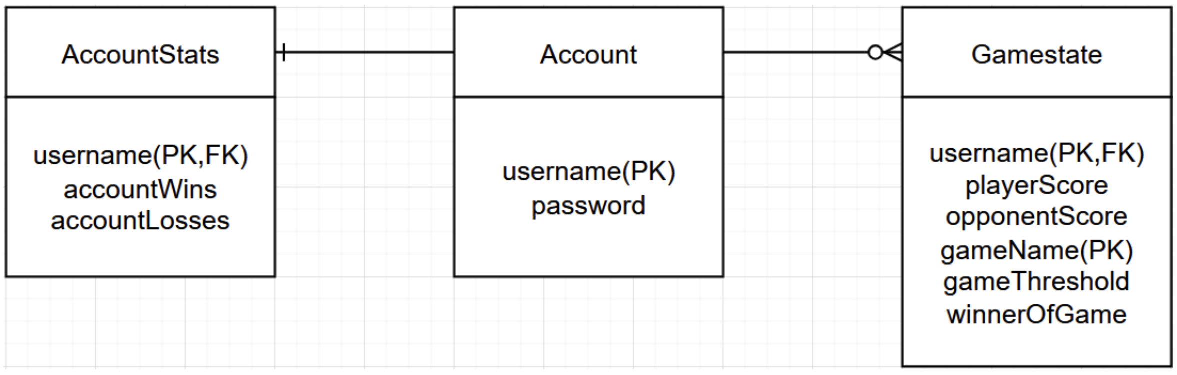
POST /subscribeToGame Subscribes the logged in user to a game

- Verb: POST
- Endpoint: /subscribeToGame
- Body: Request body is derived from the session object, which contains the username of the user.
- Responses: **201 Created** if the user subscribes to the game successfully. **200 OK** if the user is already subscribed to the game. **401 Unauthorized** if no user is logged in. **500 Internal Server Error** if issues occur with the server.

4.3 Database Design

The following section shows the relationship between the three tables that exist within the application, and are defined using the sequelize library. At a high level, these describe the way an account, as well as how the game statistics for an account is defined. Furthermore, it shows the relation between a game state and the account that is playing the game.

4.3.1 ER-Diagram



ER-Diagram depicting the database relationships.

4.3.2 Description of table relationships

In the database, there is a **one-to-one** relationship between the *Account* and the *AccountStats* table. These are the database's most tightly coupled tables, as both an Account instance and an AccountStats instance are created during the registration of an Account. The AccountStats instance has a foreign key to the account's username which is created, which creates a direct relationship between the two entities. The relationship

is defined as one which cascades on delete. The purpose of this is to ensure that if an account is ever removed, the AccountStats (and *Gamestates*) for this Account is as well. There is also a *one-to-zero-or-many* relationship between the *Account* and the *Gamestate*. This means that an Account can be associated with no gamestate, or several Gamestates. The Gamestate includes a foreign key to username, creating a direct relationship whenever a Gamestate is created. The purpose of this relationship is to allow for a player to register to a game they wish to play, and obtain a Gamestate unique for that game. Currently, the only game available is Rock Paper Scissors. The purpose of this design choice is to allow for the web application to be expanded to allow an Account to play other games as well, which the user would be able to register to, and obtain a Gamestate for. This essentially means that an Account does not need to have a gamestate, but once they decide to register for a game to play, they will obtain an entry for that game, with the possibility for an Account to have an entry for several different games.
Lastly, the password stored in the Account table is **never** stored in plain text, but is encrypted using both hashing and salting to protect sensitive information of the users. The hashing and salting occurs on the server layer, meaning the database will never know of any unencrypted passwords.

The major responsibilities/aspects of the development process has been listed below, alongside the group members that worked with

5 Responsibilities

- **Drawing design for web app**: Ali, Oscar, Jesper,
- **HTML + CSS + Bootstrap**: Ali, Oscar, Jesper, Suheib
- **Model Interfaces**: Ali, Oscar, Jesper, Suheib
- **Service Layer**: Oscar, Ali, Jesper, Suheib
- **Router Layer**: Oscar, Ali, Suheib, Jesper
- **In-Memory Service Layer Tests**: Jesper, Oscar, Ali, Suheib
- **Code revamp for user support**: Oscar, Jesper
- **Translate in-memory login to user sessions**: Jesper
- **Gameboard playing/visual logic**: Oscar, Jesper
- **Database Design**: Oscar, Jesper, Ali, Suheib
- **Final Visual re-haul**: Ali
- **Front-End testing**: Ali, Oscar
- **Validation**: Ali, Oscar, Jesper, Suheib
- **Database Service Layer Test Suites**: Oscar
- **Bug-fixing**: Oscar, Jesper, Ali, Suheib
- **Writing report**: Oscar, Jesper, Ali, Suheib

Github account names:

- Oscar Broborn - O-Brob
- Jesper Vesanen - Willard-VanDine
- Ali Aziz - A-Azi-z / Ali-Aziz-nti-johanneberg
- Mhd Suheib Shahin, - Suheib-Shahin