The Beer Game

Sprint 1

Contributors: Subigya Paudel, Korin Hoxha (Team 9)

Contents:

Introduction

On this sprint we focused on the front-end and created templates for individual pages that the server is going to deliver to the clients. We used HTML, CSS, REACT JS and SCSS. The testing was done using plain JS code. While the entire project could have been done in react, we opted out not to do that since some parts of the game, particularly the signed-in views require a lot of database access. So to better aid the integration of our work with the backend infrastructure that future teams might create, we decided to not deal with the intricacies of react routing, and rely on traditional synchronous requests to the server when the user signs in. Korin worked on the instructor view, main page and the authentication page. Subigya made a considerable work contribution and completely finished the player view. The beer game is based on supply chain and involves 4 players where each player represents one of the four stages in the chain i.e the retailer, the manufacturer, the wholesaler, the consumer. So the instructor would monitor the working of the player after each authenticates as such.

Software Requirements:

Since the pages that do not require user authentication use react, node.js, and related modules like create-react-app are required to access the templates that utilize react. The pages that use plain JavaScript, CSS and HTML, only require a web-browser. Since, the development of the templates were done in Google Chrome, Version 88.0.4324.190 (64-bit), we recommend using Google chrome whilst using this app (same version if possible) since the JavaScript engine in other browsers might differ and the behaviour of the frontend of the application might be unexpected.

Although we tried to make our pages media-responsive as much as possible we might have missed out on certain aspects, and we therefore recommend using a 13-inch device, as most of the development was done in such devices.

The landing page of the site

Home.html

The game venture shall start with the home page which follows a simplistic html design.

It has three sections:

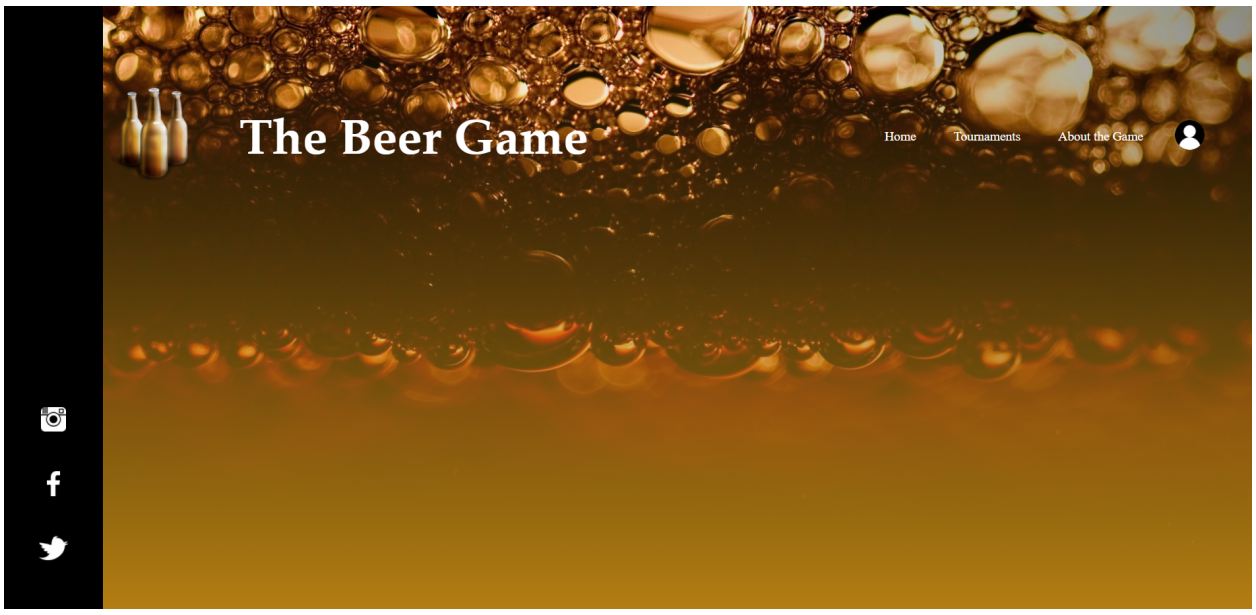
1. Section 1 is composed:

a. The logo and the name of the webpage.

b. The navigation bar with a navigation list to Home (brings the user back to the homepage), Tournaments (will later have examples of already played games), About the game (will let a first-time page visitor know more about the nature of the game: informative).

c. User logo redirects to a Log in/ Register page where a visitor can either register or login as a game player or an instructor.

d. The side bar is composed of social media links that redirect the page visitor to the webpage's social media networks.



2. Section 2:

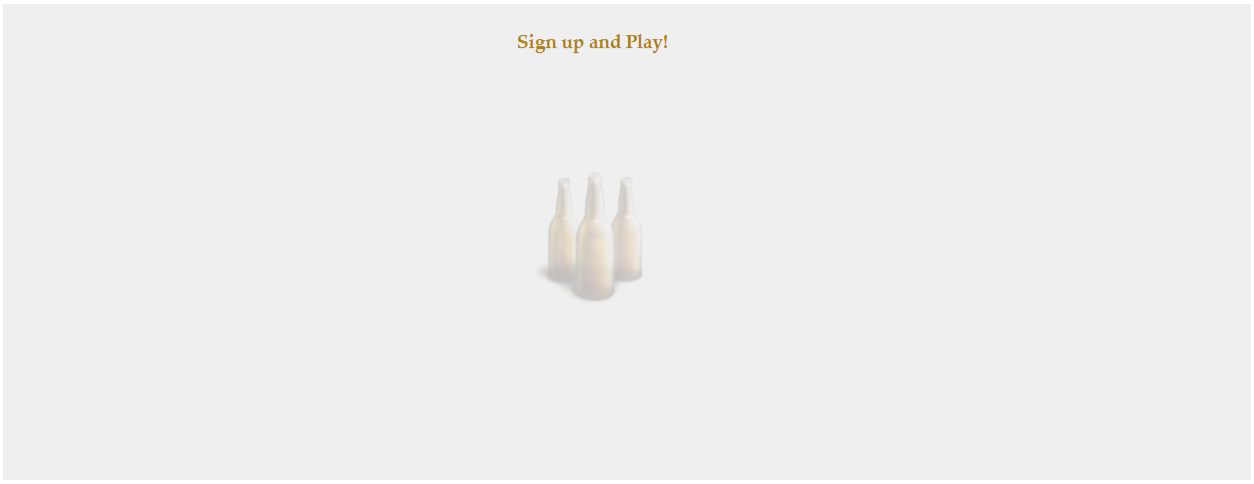
This section will contain a “service-box” composed of the 4 supply chain stages of the game, namely:

a. Retailer
b. Manufacturer
c. Wholesaler
d. Customer

Each single “service” will have a CSS styled overlay on every mouse hover.

3. Section 3:

This section is a colored footer that contains the logo with a low saturation and a motivating slogan for the first-time visitor to join the game!

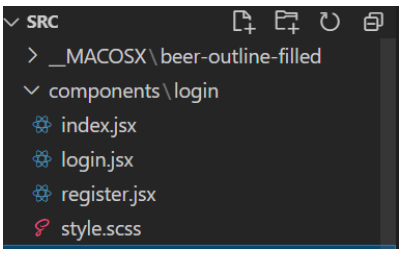


- Dominant colors:

```
#b37e13
#000
```

Login/Register page:

The page is composed of the login and register box forms which were created using React app and the styling was done using the scripting language SASS. Each form is separated in individual **.jsx** files inside a **component** folder (itself stored inside the **src** folder) so to create the special effect of hovering from one form to the other by using states as explained later on.



Login and Register boxes are going to be a separate Components so later we can easily manipulate them and decide which one to render or not depending on the application state plus it is going to make it much easier for applying some cool animation while transitioning between states.

Login.jsx:

```
export class Login extends React.Component {
  constructor(props) {
    super(props);
  }

  render() {
    return (
      <div className="base-container" ref={this.props.containerRef}>
        <div className="content">
          <div className="image">
            <img src={loginImg} />
          </div>
          <div className="form">
            <div className="form-group">
              <label htmlFor="username">Username</label>
              <input type="text" name="username" placeholder="username" />
            </div>
            <div className="form-group">
              <label htmlFor="password">Password</label>
              <input type="password" name="password" placeholder="password" />
            </div>
          </div>
        </div>
        <div className="footer">
          <button type="button" className="btn">
            Login
          </button>
        </div>
      </div>
    );
```

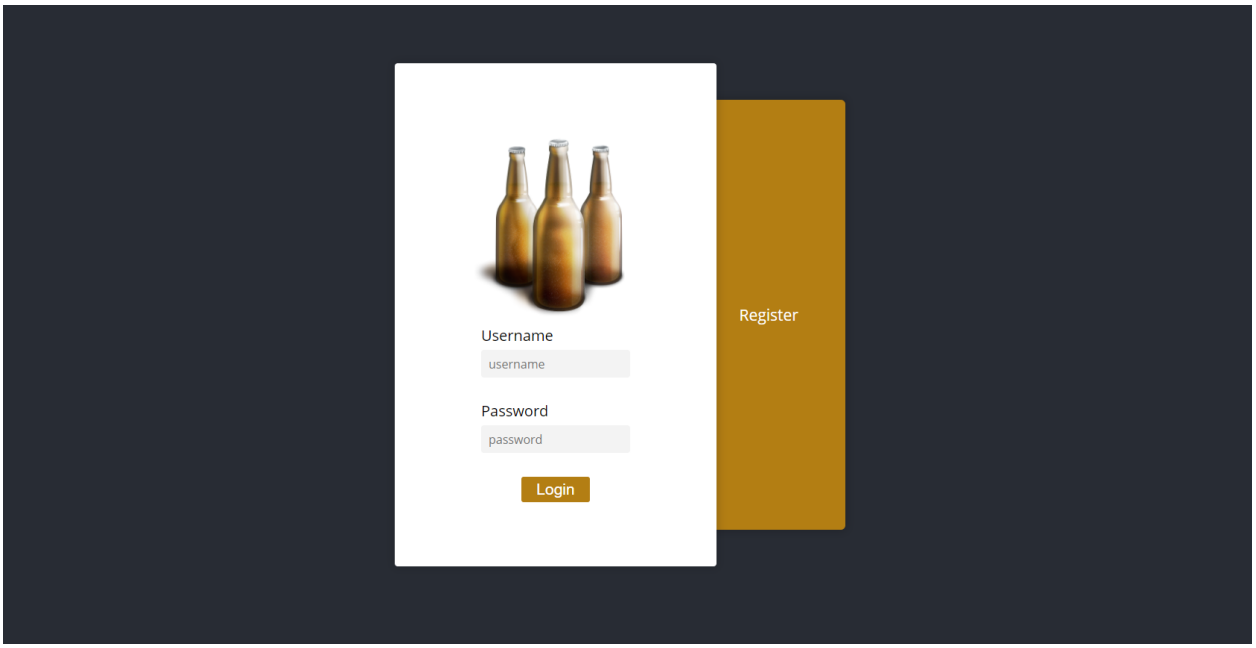
The Login component renders a form inside a container. The form will contain input fields for both the username and the password. A Login button will also be included and after implementing the backend after a correct authentication shall send the user to either a player or instructor mode page.
The style for the basic container shall be shared with the Register component in the file **Style.scss**. While the button was styled accordingly in the **App.scss** file.

Register.jsx:

This component has the exact form within a container structure, with the exception of an extra email input field.

```
export class Register extends React.Component {
  constructor(props) {
    super(props);
  }

  render() {
    return (
      <div className="base-container" ref={this.props.containerRef}>
        <div className="content">
          <div className="image">
            <img src={loginImg} />
          </div>
          <div className="form">
            <div className="form-group">
              <label htmlFor="username">Username</label>
              <input type="text" name="username" placeholder="username" />
            </div>
            <div className="form-group">
              <label htmlFor="email">Email</label>
              <input type="text" name="email" placeholder="email" />
            </div>
            <div className="form-group">
              <label htmlFor="password">Password</label>
              <input type="password" name="password" placeholder="password" />
            </div>
          </div>
        </div>
        <div className="footer">
          <button type="button" className="btn">
          <div className="footer">
            <button type="button" className="btn">
              Register
            </button>
          </div>
```



Style.scss:

As mentioned above the Login and Register components share the same style file. The base-container nests the content (which in itself nests the image i.e., logo) and the input form. Meaning the same style in the base-container of a flexbox is applied to its children to center the container box together with them. The form has a column flex-direction so the indicating label is on top of the corresponding input box. Every time the mouse hovers on the input box the box is highlighted (box-shadow) with our theme color: #b37e13.

```scss
.base-container {
  width: 100%;
  height: auto;
  display: flex;
  flex-direction: column;
  align-items: center;

  .content {
    display: flex;
    flex-direction: column;

    .image {
      width: 250px;
      margin-top: 10px;

      img {
        width: 100%;
        height: 100%;
      }
    }

    .form {
      margin-top: auto;
      margin-bottom: auto;
      display: flex;
      flex-direction: column;
      align-items: center;

      .form-group {
        display: flex;
        flex-direction: column;
        align-items: flex-start;
        width: fit-content;
        label {
          font-size: 20px;
        }
        input {
          margin-top: 6px;
          min-width: 2px;
          height: 37px;
          padding: 0px 10px;
          font-size: 16px;
          font-family: "Open Sans", sans-serif;
          background-color: #f3f3f3;
          border: 0;
          border-radius: 4px;
          margin-bottom: 31px;
          transition: all 250ms ease-in-out;
          &:hover {
            background-color: #ffffff;
            box-shadow: 0px 0px 14px 0.3px #b37e13;
          }

          &:focus {
            outline: none;
            box-shadow: 0px 0px 12px 0.8px #b37e13;
          }
        }
      }
    }
  }

  .footer {
    margin-top: 1px;
    margin-bottom: auto;
  }
}
```

Index.jsx:

For the Login and Register components to be easily imported into the main **App.jsx** file the corresponding files have been exported firstly in the **index.jsx** file.
With the usage of this file the style from **style.scss** is also applied on **login.jsx** and **register.jsx**.

components > login > index.jsx

```
1   import "./style.scss";
2
3   export { Login } from "./login.jsx";
4   export { Register } from "./register.jsx";
```

App.jsx:

This file will render the *Login* and *Register* components (imported from **index.jsx**) mentioned above within the same page while applying a mouse transition through the usage of the state *isLogginActive* to see if *Login's* render is active and *onComponentWillMount* hook with a right (later used) default value. If *Login* is not rendered, then *Register* render is active. A transition animation between the two components is facilitated by referencing a *div* container.

In order to transition we create a functional component "*RightSide*" (line 58-70). This component is referenced by *containerRef* (line 49) on the App class. Whether we want to transition to Login or Register state, that is done by passing the property object argument that holds either one component (line 31-56). As mentioned above the component selected is detected on the active state of either one by the True (Login component is active) or False (Register component is active) value of the state *isLogginActive*. To detect the mouse event on the "*RightSide*" component the event *onClick* is passed inside this component. The method changeState handles the click on that component. This method itself is binded to the *onClick* event (line 50).

When one component (*Login*) or the other (*Register*) is active by the mouse click event the corresponding style is applied from **App.scss** by the usage of the custom classes (*add*, *remove*) on the "*RightSide*" component.

```jsx
import React from "react";
import "./App.scss";
import { Login, Register } from "./components/login/index.jsx";

class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      isLogginActive: true
    };
  }

  componentDidMount() {
    this.rightSide.classList.add("right");
  }

  changeState() {
    const { isLogginActive } = this.state;

    if (isLogginActive) {
      this.rightSide.classList.remove("right");
      this.rightSide.classList.add("left");
    } else {
      this.rightSide.classList.remove("left");
      this.rightSide.classList.add("right");
    }
    this.setState(prevState => ({ isLogginActive: !prevState.isLogginActive }));
  }

  render() {
    const { isLogginActive } = this.state;
    const current = isLogginActive ? "Register" : "Login";
    const currentActive = isLogginActive ? "login" : "register";
    return (
      <div className="App">
        <div className="login">
          <div className="container" ref={ref => (this.container = ref)}>
            {isLogginActive && (
              <Login containerRef={ref => (this.current = ref)} />
            )}
            {!isLogginActive && (
              <Register containerRef={ref => (this.current = ref)} />
            )}
          </div>
          <RightSide
            current={current}
            currentActive={currentActive}
            containerRef={ref => (this.rightSide = ref)}
            onClick={this.changeState.bind(this)}
          />
        </div>
      </div>
    );
  }
}
const RightSide = props => {
  return (
    <div
      className="right-side"
      ref={props.containerRef}
      onClick={props.onClick}
    >
      <div className="inner-container">
        <div className="text">{props.current}</div>
      </div>
    </div>
  );
};

export default App;
```

In the login container we specify attributes that help with the transition animation. The z-index of 1 specifies the stack order on top as the active container. The active box *container* is nested into the *login* container. It will have a white background.

The *right-side* container is the non-active container and will be colored #b37e13 accordingly and positioned horizontally on the right at-34% below the active *container* which in contrast is colored white when active as mentioned above.

The transition animation is applied to the *right* child of the *right-side* component (if not active) with an offset of 40% and with a hover of the mouse to 45%. It will also have a white colored text indicating the non-active component. The same reasoning for the *left* child of the *right-side* component if non-active.

App.test.js:

Test cases run on the authentication page through the file App.test.js:

1. Renders the whole page with no errors.
2. App renders the login button text
3. App has a password placeholders
4. App has a username placeholders
5. Invalid password length is detected
6. Invalid email was detected

Signed In views

The Instructor View:

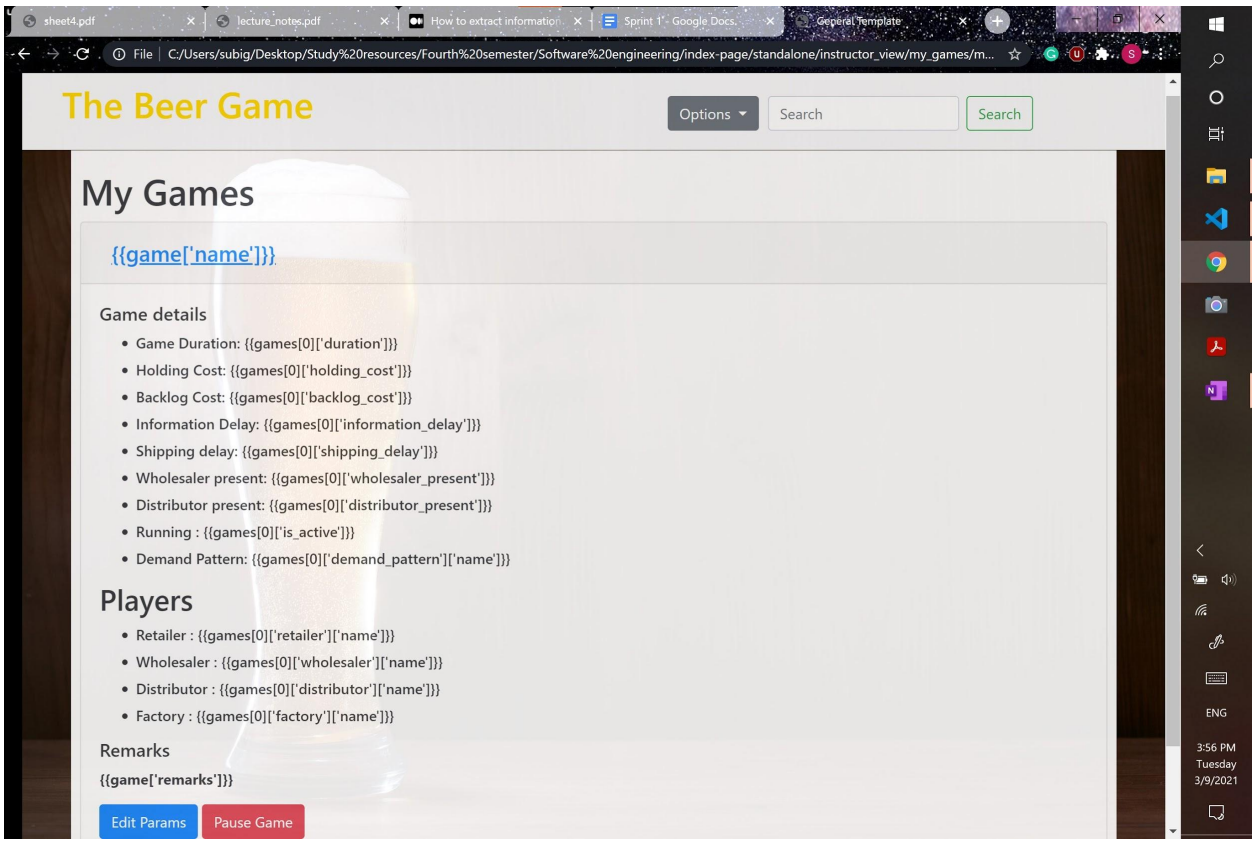
The instructor view consists of the templates to be used when the instructor logs in to our site. The bundle consists of four templates, namely:

1. **The my games page** (which is the landing page for the instructor once s/he is signed in)
2. **The edit games page** (which allows the user to edit games if they aren't running)
3. **The create games page** (which allows the user to create new games)

We utilized bootstrap 4 and JavaScript to give the pages better looks and functionalities like input validation.

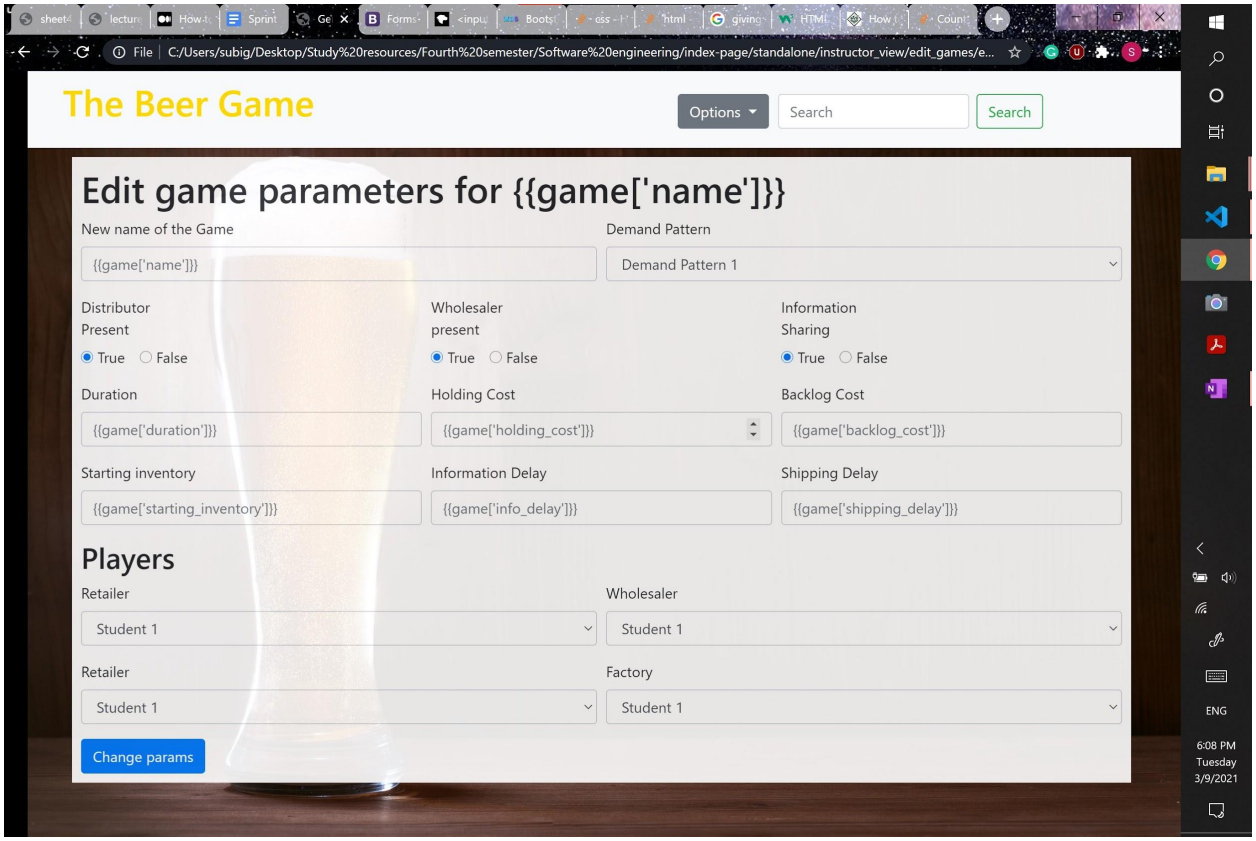
Detailed explanation of the templates:

The my games page



The above page shows the my games page for the instructor. The main content of this page is the accordion in the middle of the page where the instructor can see all the games that s/he is monitoring. Furthermore, the instructor can also edit the game parameters from this page and pause the game if the game is running. Since displaying the two buttons requires us to check if the game is running or not, it would be best if conditional rendering is used here. To that end, Jinja templates supported by Django are also placed in the code which will see if the game is running or not and will place the buttons accordingly. The two cards in the accordion are just placed there to showcase the intended functionality.

The Edit games page



The above picture shows how the edit game page looks like. We have used bootstrap 4 forms and classes to make the form look better, and to also use the built in validation tools. Furthermore, we also used a function called validation( ) to check if two or more roles are being assigned to the same player. As is evident from the above picture, we also placed django's jinja templates to assign placeholders to the form and also to get the name of the game as well. However, the form also intends to use such jinja templates extensively since the code hints at using jinja templates to render the various options in the select tags, such as the demand pattern and the students. The custom-made validation() function is as follows:

```
validation = () => {

  let playerslist = {};

  document.querySelectorAll('.players').forEach((element) => {

    if(playerslist[element.value] == undefined){

      playerslist[element.value] = 1;

    }else{

      playerslist[element.value]++;

    }

  })

  for(item in playerslist){

    if(playerslist[item] > 1){

      console.log(playerslist[item]);

      alert('All player selections must be different')

      return false;

    }

  }

  alert('Are you sure that you want to proceed?');

  return true;

}
```

The create game form is very similar to the edit game form, and the only differences are between the placeholders.

The Player View:

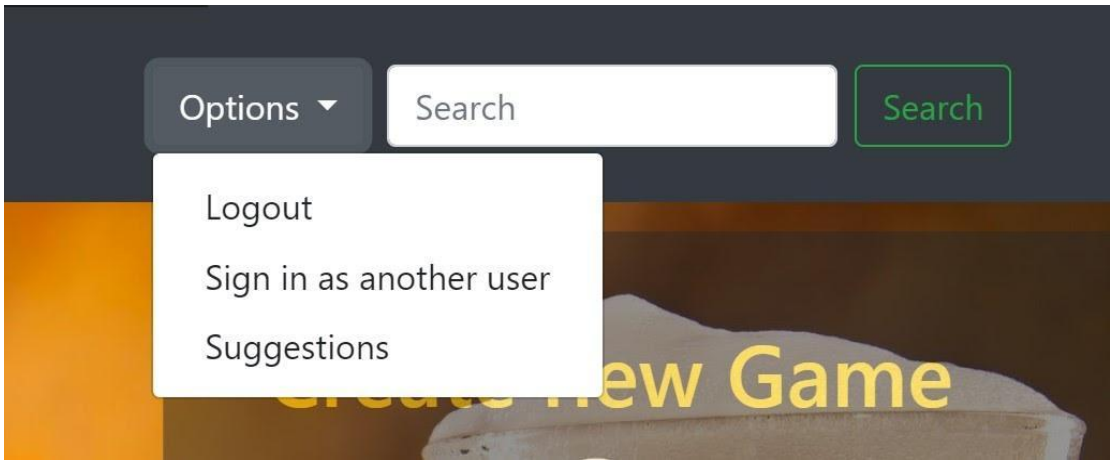
The player view bundle has web_pages pages, each with its own CSS, HTML and JavaScript files. Each of these pages is adapted to a specialized situation. We did not choose to use react here, since frequent communication with the server is essential when players play the game and dealing with react routing would have made the code less readable. So in order to make the future implementations easier, we chose to use plain HTML, CSS and JavaScript for this section. They are as follows:

1. **index_page**: This is the home page, and players are redirected to this page when they sign in
2. **my_games**: This is the page that players are redirected to when players click on the My games link on the side bar.
3. **my_instructors**: This is the page that players are redirected to when they click on the My instructors link on the side bar.
4. **single_game**: This is a page that contains the template for showing all information pertaining to a single game. Players must be redirected to this page utilizing this template if they click on a link that is supposed to show them information pertaining to a single page.
5. **game_screen**: This is a page where players will play the game.
6. **general_template**: A template that can be utilized for showcasing additional functionalities, that has not been covered by the plan, like customizable game settings. Furthermore, it can also be used to implement pages that are trivial to build, like the one containing data policy of the game.

If one sees the source code/rendering of these pages, they can observe Jinja templates({{}} and {%%}). This was done with the assumption that a user will use Django to transfer information from the backend to the frontend, but the code can be easily adapted to PHP and python cgi scripts as well.

Detailed explanation of the templates:

INDEX PAGE:

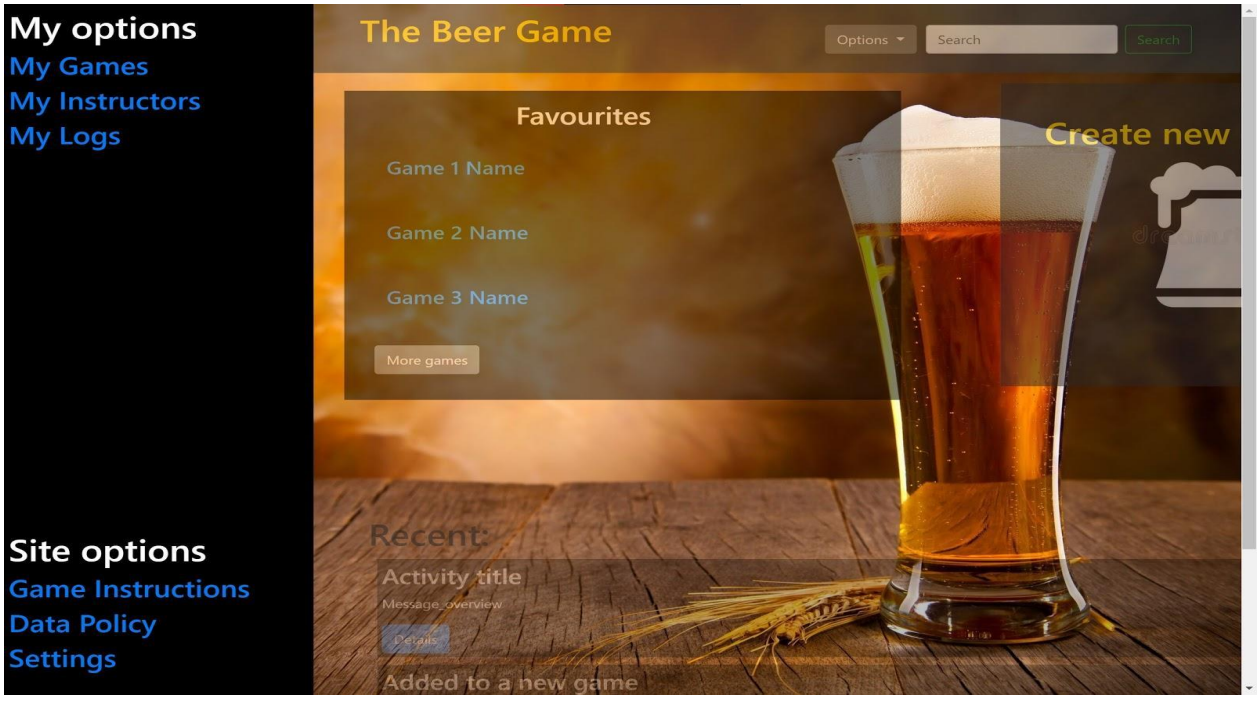


First, let us start with the utilities in the navbar. These utilities consist of a dropdown labeled Options, and the search bar. The options dropdown provides the player with the means to logout, sign in as another user, and to provide suggestions to improve the website (not included in the project plan). The search bar can be used to perform a general search across all the tables in the database pertaining to the user and to show relevant class instances. Since this process is highly dependent on the backend, the frontend of this search bar has not been implemented. Both the dropdown and the front end have been implemented using bootstrap 5.0, and the code to the entire navbar which contains these utilities can be found in the code below.

```
<nav class="navbar navbar-expand-lg navbar-light bg-dark fixed-top side_shift" id="navbar">
  <div id='logo'>
    <a class="navbar-brand" href="">
      <h1>The Beer Game</h1>
    </a>
  </div>
  <div id='utilities'>
    <form class="form-inline my-2 my-lg-0" id='search'>
      <input class="form-control mr-sm-2" type="search" placeholder="Search" aria-label="Search" />
      <button class="btn btn-outline-success my-2 my-sm-0" type="submit">Search</button>
    </form>
    <div class="dropdown" id='options' onMouseLeave='cleanup'>
```

```
      <button class="btn btn-secondary dropdown-toggle" type="button" id="dropdownMenuButton"
data-toggle="dropdown"
        aria-haspopup="true" aria-expanded="false">
        Options
      </button>
      <div class="dropdown-menu" id='options_drop_down_menu' aria-labelledby="dropdownMenuButton">
        <a class="dropdown-item" href="#">Logout</a>
        <a class="dropdown-item" href="#">Sign in as another user</a>
        <a class="dropdown-item" href="#">Suggestions</a>
      </div>
    </div>
  </div>
 </nav>
```

The Sidebar



In the figure above we can see the sidebar, a feature that is present in all web pages that the players can access once they are logged in. In all the sidebars except for the one in the game_screen template, there are my options, and site options.

My options consist of my games (which redirects the players to a page that displays all the games that the players can join), my instructors (which redirects the players to a page showing all the instructors that the player is associated to), my logs (though not in the plan, it an interface which shows the player's activity history on the software).

Site options consists of Game instructions (which redirects the user to a site which explains what one has to do in the beer game), data policy(which shows how the users' data is used), and settings (this isn't in the plan but this is a place where the clients can change their preference for the website, like themes and background).

The sidebar opens up when the user hovers over the thin gray div which is sticking to the left of the page. The HTML, code for the sidebar is as follows:

```
<div id='sidebar' onmouseenter='openNav()' onmouseleave="closeNav()">
  <div id='container_user_options' class='sidebar_options'>
    <h1 class='sidebar_options'>My options</h1>
    <a href="">
      <h2 class='sidebar_options'>My Games</h2>
    </a>
    <a href="">
      <h2 class='sidebar_options'>My Instructors</h2>
    </a>
    <a href="">
      <h2 class='sidebar_options'>My Logs</h2>
    </a>
  </div>
  <div id='container_site_options' class='sidebar_options'>
    <h1 class='sidebar_options'>Site options</h1>
    <a href="" onclick="game_instructions()">
      <h2 class='sidebar_options'>Game Instructions</h2>
    </a>
    <a href="">
```

```
      <h2 class='sidebar_options'>Data Policy</h2>
    </a>
    <a href="">
      <h2 class='sidebar_options'>Settings</h2>
    </a>
  </div>
</div>
```

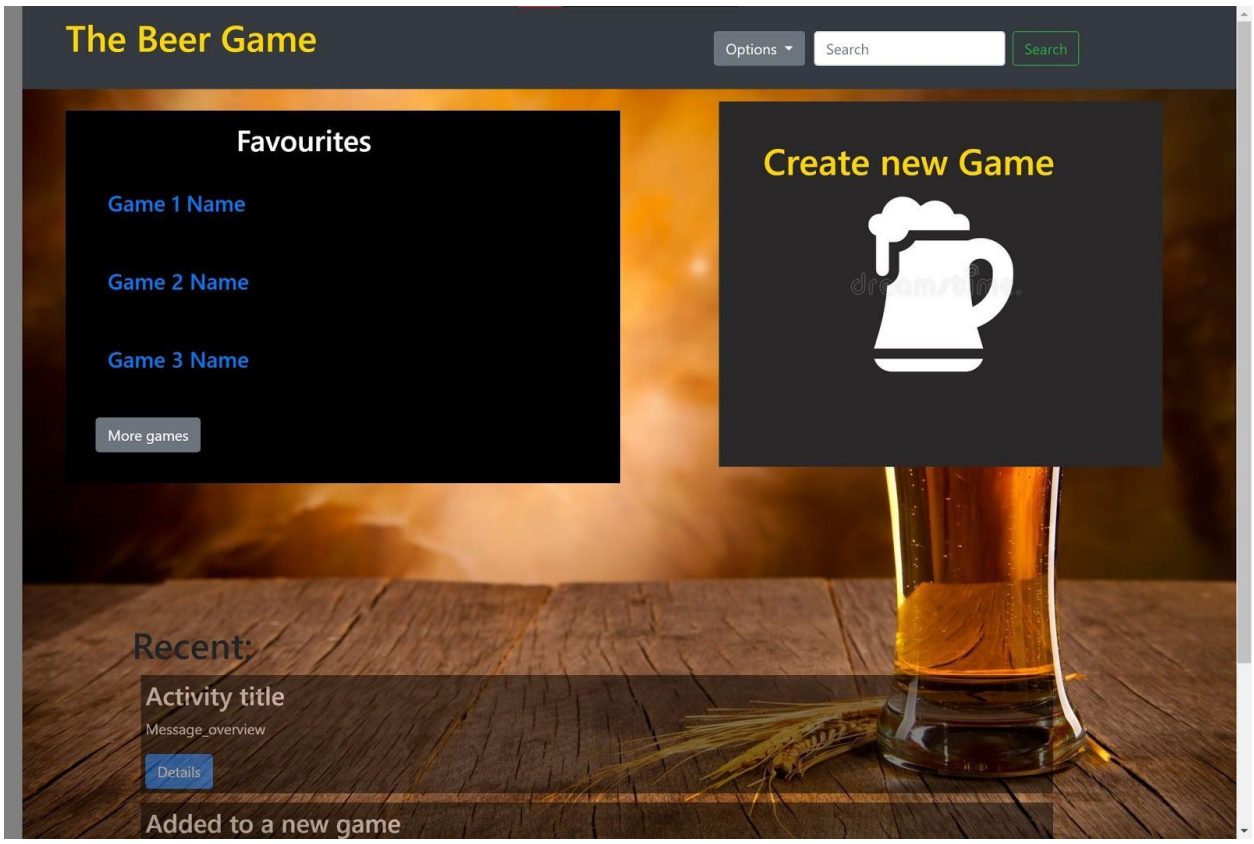
As we can see from the code above, the opening and closing of the navbar is controlled by the event handlers openNav() and closeNav(). The code to these event handlers are as follows:

```
function openNav() {
 console.log("hello");
 var sidebar = document.getElementById("sidebar");
 sidebar.style.animation = "expand";
 sidebar.style.animationDuration = "0.6s";
 sidebar.style.animationDirection = "normal";
 sidebar.style.animationIterationCount = "1";
 sidebar.style.animationPlayState = "running";
 document.querySelectorAll(".side_shift").forEach((element) => {
  element.style.animation = "shift_left";
  element.style.animationDuration = "0.3s";
  element.style.animationDirection = "normal";
  element.style.animationIterationCount = "1";
  element.style.animationPlayState = "running";
  element.onanimationend = (event) => {
    event.target.style.marginLeft = "25%";
    event.target.style.opacity = "0.5";
  };
 });
 sidebar.onanimationend = (event) => {
```

```
    event.target.style.width = "25%";
    event.target.style.backgroundColor = "black";
  };
  document.querySelectorAll(".sidebar_options").forEach((element) => {
    element.style.visibility = "visible";
  });
}

function closeNav() {
  var sidebar = document.getElementById("sidebar");
  sidebar.style.animation = "shrink";
  sidebar.style.animationDuration = "0.2s";
  sidebar.style.animationDirection = "normal";
  sidebar.style.animationIterationCount = "1";
  document.querySelectorAll(".side_shift").forEach((element) => {
    element.style.animation = "shift_back";
    element.style.animationDuration = "0.1s";
    element.style.animationDirection = "normal";
    element.style.animationIterationCount = "1";
    element.style.animationPlayState = "running";
    element.onanimationend = (event) => {
      event.target.style.marginLeft = "0%";
      event.target.style.opacity = "1";
    };
  });
  sidebar.style.animationPlayState = "running";
  sidebar.onanimationend = (event) => {
    event.target.style.width = "20px";
    event.target.style.backgroundColor = "grey";
  };
  document.querySelectorAll(".sidebar_options").forEach((element) => {
    element.style.visibility = "hidden";
  });
}
```

As we can see here, the main elements of the webpage which aren't the part of the sidebar are given the class name side_shift. So when the openNav function runs, these elements' 'left margin' property is increased, making them shift to the right. At the same time, the width of the sidebar is increasing, and the visibility of the elements inside it is being set to 'visible'. CSS animations are used to make these transitions smoother. The exact process is reversed in closeNav, and we see the observed effect.



Here we can see the contents of the index page. The favourites panel on the top half of the page is a bootstrap 5 accordion and all the game details pop up when the user clicks on the game name. They also automatically 'tuck themselves' when the mouse leaves the favourites panel. To visit all the games that one can play, they must simply click the more games button on the favourites panel.

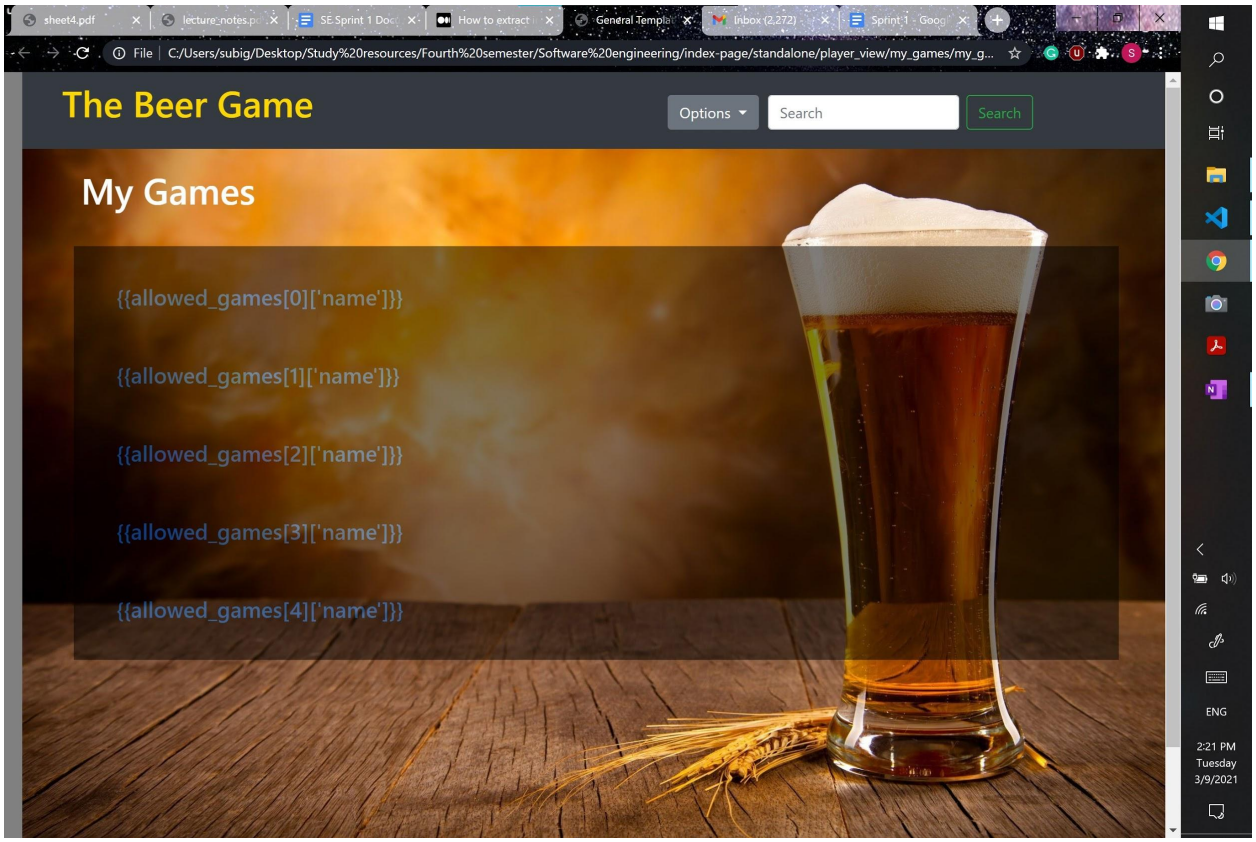
In the bottom-half of the page, we can see the recent section, which consists of recent developments that pertains to the user. One can click the details button and then they must be redirected to a separate page which shows all the details about this development.

All the elements in this page light up when a mouse hovers over them and dim when the mouse leaves the element.

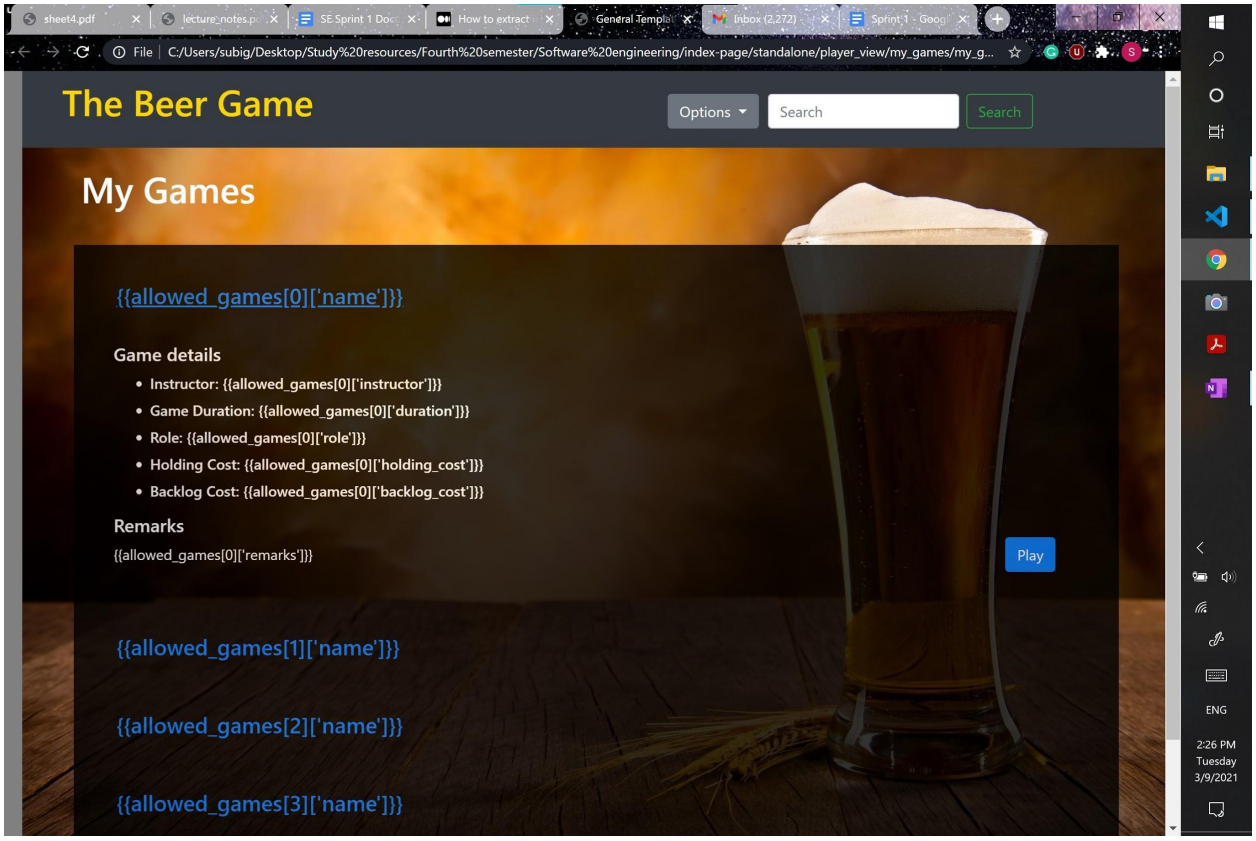
The HTML code of this entire section can be found as child elements of the div whose id is contents. The javascript functions handling the brightness changes for the accordion are respond() and cleanup(), and that for the create game icon is glow() and dim().

The code also contains sections that dynamically render content if django is used as the backend.

My games:

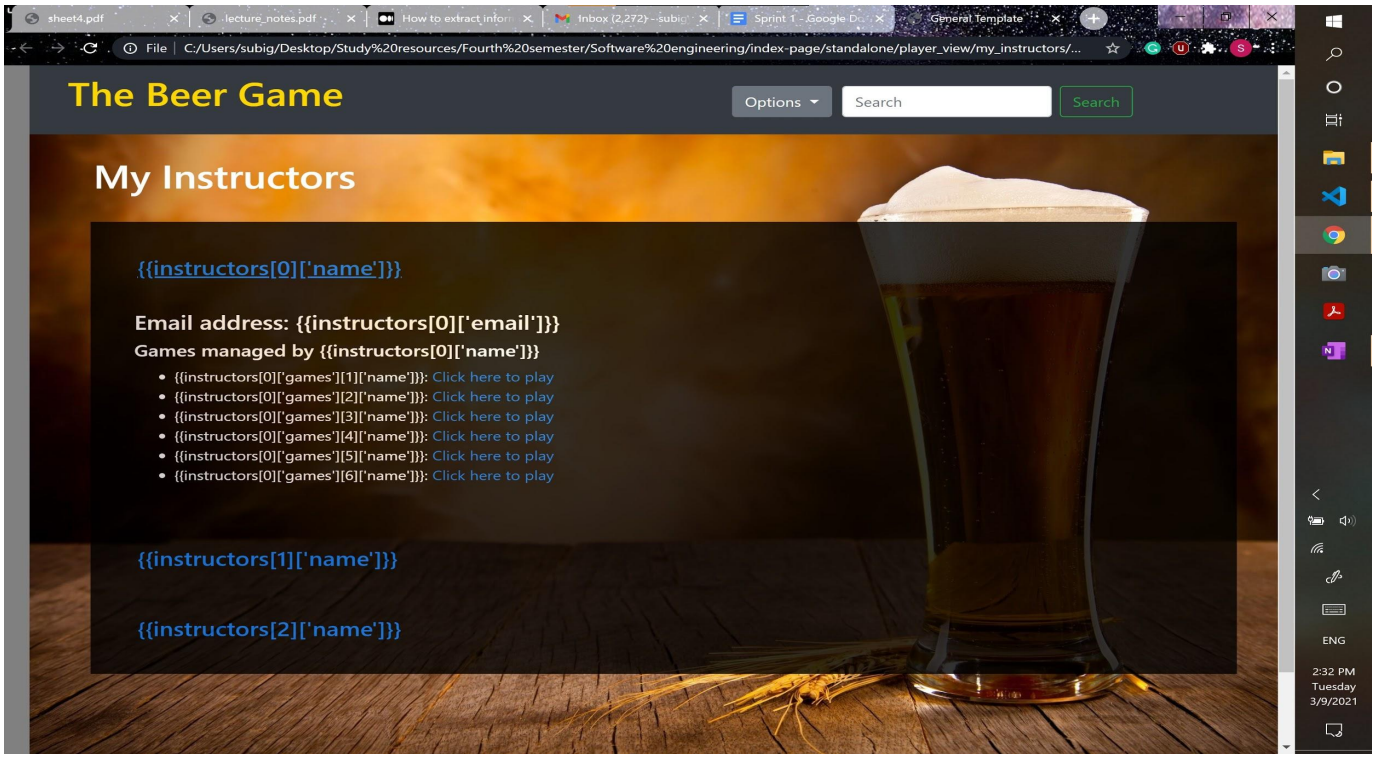


The above screen in the view of the my games screen. It has the usual navbar on the top of the page and the sidebar on the left side of the page to provide site and personalized options. The main content of this page is a bootstrap accordion which lists all the games that the player is eligible to play.



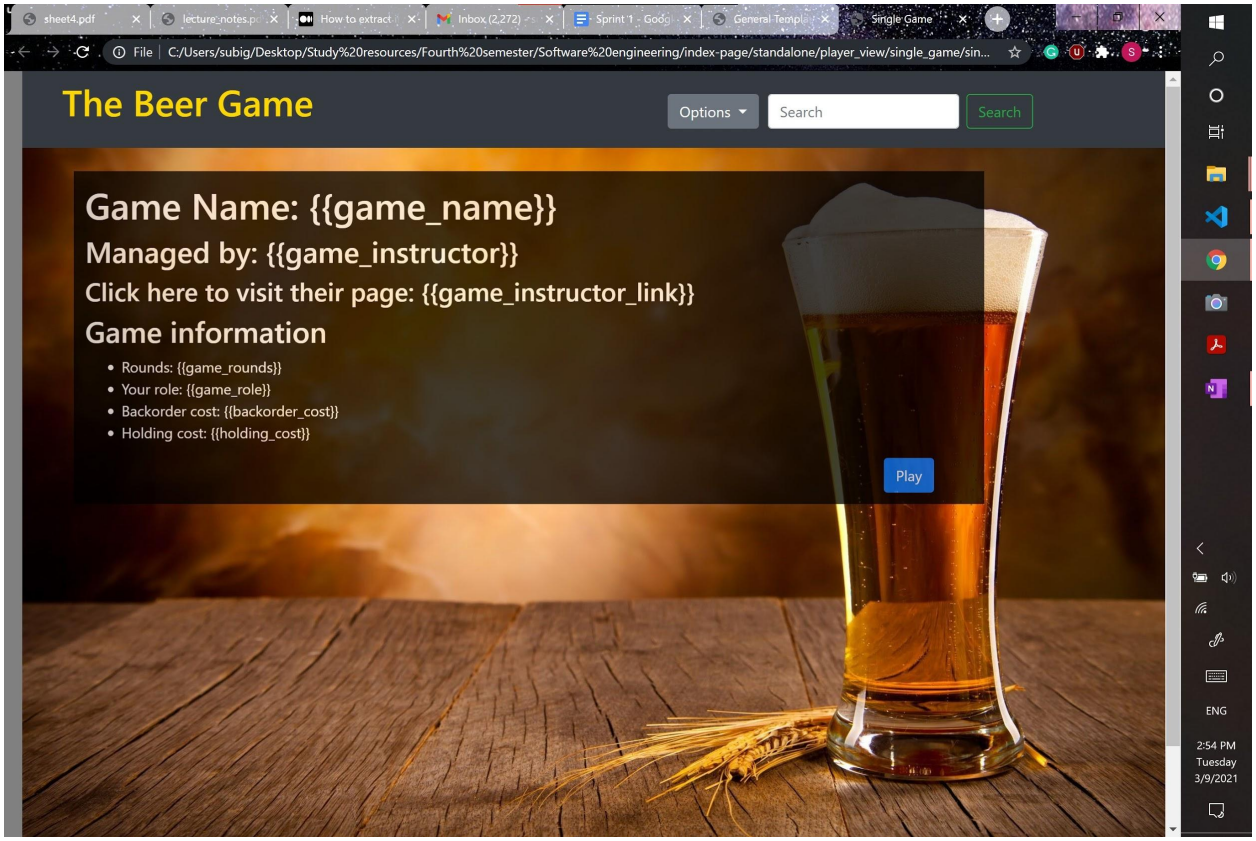
As shown above, players need to only click the game in the accordion to be presented with all the game-related details and a button that will lead them to play the game.

My instructors:



As we can see above, the my instructors page contains the navbar and side panel similar to that of the index page. It also, like the my games page, has a bootstrap accordion in the middle of the page which displays all the instructors that the player is associated with. A player only needs to click the accordion to be presented with the email address of the instructor, and the games that the instructor manages which the player is eligible to play. Along with the names of those games, the links to individual pages of the games have also been provided.

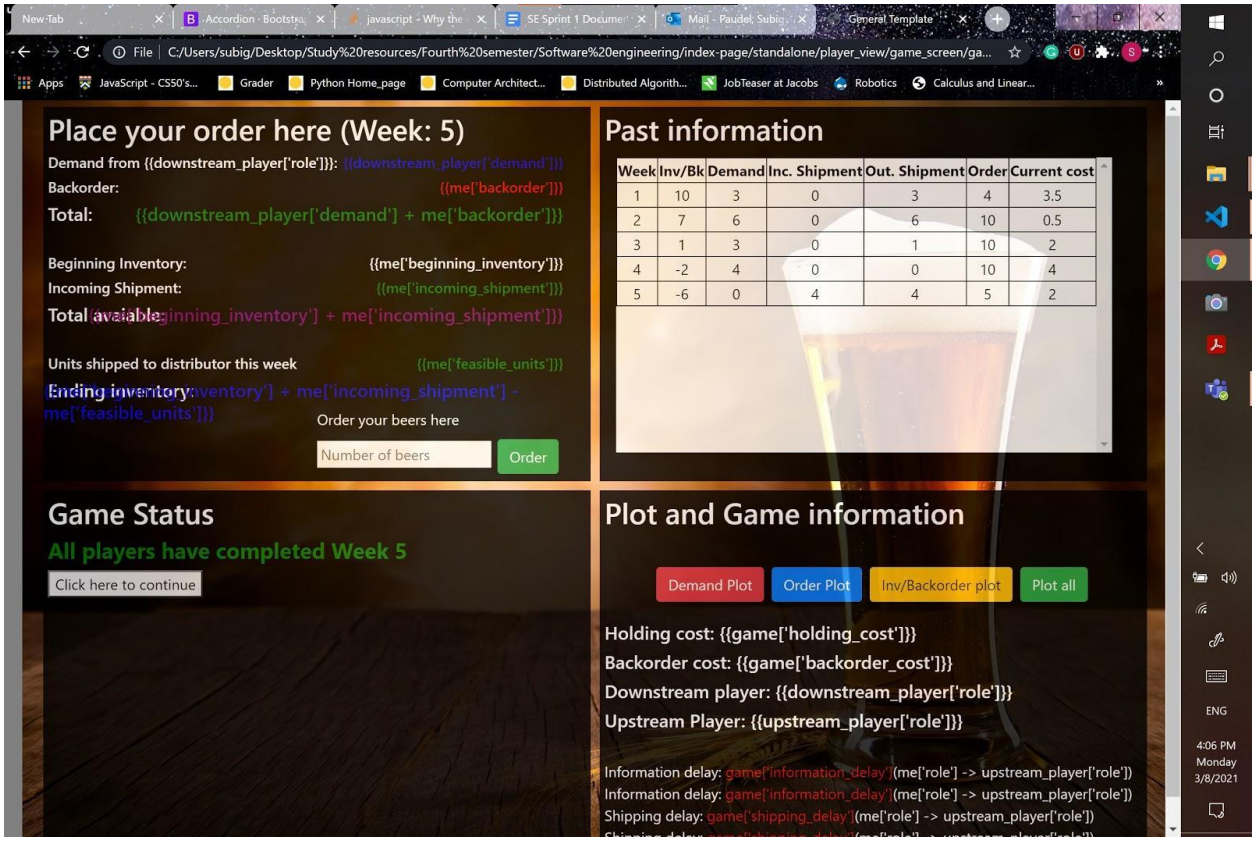
Single page:



The above page is designed for showing the game information for a single game. It shows all the game parameters of the game, and the role designated to the player viewing this game. Furthermore, it also shows the instructor who is managing this game and gives link to visit their page as well.

The game screen:

The game screen bundle is located in the game_screen directory, and has the files, game_screen.html, game_screen.js, and game_scrren.css.

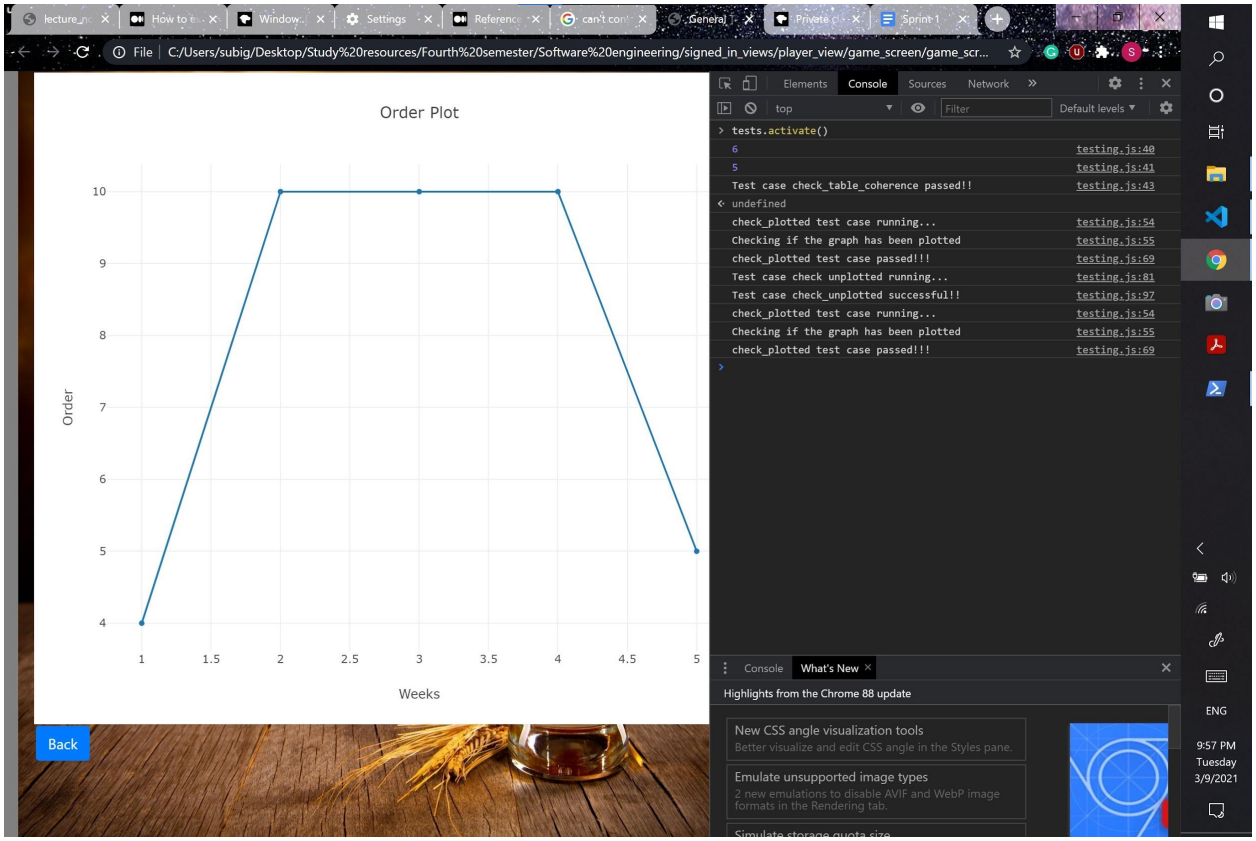


The above is the general game screen for players regardless of their roles. The game screen had been divided into four quadrants. They are as follows:

1. The first quadrant (the upper left quadrant): Is the place where players can see their current status (inventory,existing backorder, the shipment that they will receive this week), as well as the current demand and backorder. It is also from where they can order their beers for the current week.
2. The second quadrant (the upper right quadrant): Is the place where players can see their past details. Given that there may be tens of weeks here, the table has been made scrollable so that the table contents can be viewed without them interfering with other HTML elements.

3. The third quadrant (the lower right quadrant): Is the place where a player can see if other players in the game have submitted their orders for the week. The content of this page is made such that it refreshes every 15 seconds. Since there need to be backend scripts that will provide a response to an asynchronous request made by the function countdown(), this section either shows that the game is ready to proceed after 15 seconds, or the timer within this section is reset.
4. The fourth quadrant (the lower right quadrant): provides the user with an option to plot graphs, and see the game-wide details like the backlog cost, holding cost and who the players' upstream and downstream players are. There are four types of graphs that the player can plot. They are demand plot handled by the plot_demand() function, order plot handled by the plot_order() function, inventory plot handled by the plot_inventory() function, and plot all handled by the plot_all() function. All these functions, except the plot_all() function, just extract the data out of the table in the second quadrant and delegate the actual function to plotting the graph to the function plot(). Plot uses the Plotly library to plot the graph, and at the same time it also creates a div and hides all other quadrants to give us a page-wide graph. Upon the display of this page-wide graph, a button is also created to go back to the normal view of the page.

TESTS



Since most of our code was written in plain HTML, JavaScript, and CSS, we had to create our own means of testing whether certain components were rendered, removed or were visible after certain action was performed by the user. Since no part of the frontend was as user-interactive as the view in which people played the game, we decided to concentrate our test cases here. Along with the required javascript files, a testing.js file has also been linked to this page which provides us with the means to perform testing. Since the ability to plot graphs is the most complicated mechanism in this page, we decided to largely focus our testing on that. Our testing covers testing whether the graph was plotted when one of the buttons were pressed and if the graph is removed when the back button under the graph was pressed. Furthermore, we also test if the data in the table is coherent to the current week that the player is playing for. One can easily switch to the testing mode of this page by running the command tests.activate(). If this command is not run before testing, then no other commands related to the testing will work, since testing here requires us to place appropriate event handlers.

References:

https://getbootstrap.com/docs/4.0/getting-started/introduction/

https://www.w3schools.com/

https://www.faculty.jacobs-university.de/pbaumann/iu-bremen.de_pbaumann/Courses/SoftwareEngineering/index.php#mat