CSC 667 – Term Project

Exploding Kittens

Team Members

Adam Belaid

Jo Ey Chong

Rémi Teisseyre

Pierre Antoine

Contents

Github Repository

https://github.com/sfsu-csc-667-spring-2021-roberts/term-project-team-ajrp

Heroku Link

https://csc667-ajrp.herokuapp.com/

Project Introduction

This is a term project in which we are tasked to select and create a simply online multiplayer card game. The game in this project would be Exploding Kittens. The goal of this project is to design, plan and implement a functional and not-so-bad looking web app that will be deployed in Heroku, the link is provided above. Do note that accessing Heroku might take awhile as it loads up from disk, the is a limitation of the Heroku's free tier.

Local Execution

To run the game on local machine, you would first need to:

`git clone https://github.com/sfsu-csc-667-spring-2021-roberts/term-project-team-ajrp.git`

Make sure you are in the root of the cloned folder, then create or link an `'.env'` file which includes your local machine database's URL assigned to `DATABSE_URL` run:

```
npm install
```

```
npm db:migrate
```

```
npm run start
```

Go to a browser and enter `localhost:3000`, the web app should run.

Assumptions

The project is built from scratch and contains routes that link users to specific areas we want them to, the frontend user interface clearly guides and tell users where they should go. We do not claim 100% bug free and solid security, especially when the users explicitly program on the browser's console or entering URL where they shouldn't have been into.

Pre-implementation

Before actual implementations, we design and plan out it's database fields and tables, the system designs, and the user interface design. Along the way, we discover possible frameworks and libraries that help us ease the system implementations a little. Here are the list of technologies, frameworks, or libraries that we used in this game:

- Node.js
- Express.js
- Express-session
- Postgres
- Pg-promise
- Sequelize
- Passport.js
- Bcrypt
- Socket.io
- Pug view engine

Each of these technologies' usage will be further explained below.

Implementation

Folder Structures

Though we add, change and deletes multiple files and folders, the folder structure still mostly remains the same since the start when we first used express generator to quickly generate a folder structure for our project. As we added more things, we have to make it more obvious and organized to easily allow us to understand what each of us have worked on.

- The `config` folder contains configuration for our database connection, passport.js strategy and configurations, and our socket.io configurations.
- `db` folder contains our pg-promise implementations that allow us to get data back from connected database.
- `migrations` folder contains database migrations, implemented with Sequelize. This allows us to easily get updated on recent works of fields and tables, or when a new table is created and needs to be added into local machine or on Heroku.
- `public` folder contains files that are accessible by the public, such as stylesheets, images, and scripts that needs to be connected by HTML.
- `routes` folder contains our express routes that receive requests and send responses back to client. There are also logics that process incoming data. For organization, the folder has subfolders that will be accessed depending on whether a user/request is authenticated.
- `views` folder contains our pug files, which helps dynamically render HTML and send back to clients.
- On the root of the project folder itself contains an `app.js` file that initialize our express application, sessions and routes.

Technologies Implementations

Authentication

The game contains simple login and register pages that allow users to register and log in. There are simple logics that checks for errors during login or register processes, such as taken username, password don't match, no user exists, etc. Passwords are only stored in the database after being encrypted by `bcrypt`. If a user tries to log in, the entered password will be encrypted and compare with the encrypted password in the database.

The game wouldn't be useful if it doesn't store user sessions. With sessions, we can see who is logged in and proceed to make routes and logic to create a personal space for the user. It also allows our users sign in once instead of everytime they try to access the game.

In this project, to store the sessions, we used `express-session` and `passport.js`. By default, the sessions will be stored in memory of the server, and with Heroku free tier's limitations, this means that the sessions on the server will be destroyed everytime Heroku's instance get shuts down, and users would have to log in again. To avoid this, we could use a session store and store it in the database. After further discussion, we didn't think it's a priority and local development didn't show much of a drag for us, so we just left it by default. If a session is detected from the backend, the user will always be directed to authenticated pages (not homepage with login/register).

`Passport.js` contains strategies that allows us to do authentications, as we need only a simple one, we used local strategy and defined our own logic that passes in login data from client, searches for match in the database, compares the encrypted password and returns a user object that get can be then used to store in our session. `Passport.js` allows us to use flash to specify messages that can be used to alert users on the frontend, but unfortunately, we did not get that working.

Dashboard

After authenticated, user is directed to a dashboard page. The dashboard page allows user to logout, create lobby, or join a lobby from the list of lobbies.

Chat

The chat is done through the socket io middleware. The socket has a server side that is initialized when the server is launched. For the client side, it is initialized when the user enters in specific pages. The server side has multiple endpoints to redirect emitted messages to their correct destination. In order to do this, we used rooms, and every time a user enter a chat page, he automatically provides a key to that room. This key is used to link players that are on the same pages and facilitate communication.

Lobby

When a user creates a lobby, the list is updated in the database and the user is automatically redirected to the lobby page. Each lobby page is unique. The lobby will wait for enough players to start a game. A chat will also appear on the right of the page, this allows users in the same lobby to chat to each other and contents are updated in real time.

This is possible with `socket.io`. The start button creates the game and through the socket io room, redirects all users to the correct game page, where they will be put into new socket rooms for the game, and the game can start. The change lobby function allows for a change in the name of the lobby, which will show in the dashboard for all players to see. The exit button redirects the user to the dashboard to see other lobbies that are active.

Game

The game is also done through the socket io middleware. After the initial page is setup, having, initialized the game and cards in the database, and having fetched the first cards from the database, the player's movements are shared through the socket to the other players with the room key that was initialized at game creation. From there, each card type has its own socket endpoint that is emitted when the player plays a card, and is acted upon when the move is received. This enables for real time feedback. The socket io message received by the players notifies them to pull the updated data from the database, where the actual game state is stored. The drawback is that in order to keep track or who owns what card, there are constant read write commands to the database, which in certain situations may result in a 'laggy' experience. The player is also allowed to leave the game if he wishes to do so, which will throw away all his cards and redirect him to the dashboard.

HTML, Scripts, CSS

Pug view engine renders the page for us, we can also pass in data so it renders slightly differently on different scenario.

Some pages also have scripts that allow us to manipulate the interface without having the users to refresh their page for update. A solid example would be the chat in lobby, where contents from other players are shown in real time by appending list without having the users to refresh their page.

CSS are pure and written without frameworks such as Bootstrap. With some time, the pages look a lot more nice looking than without it.

Difficulties and Challenges

Jo Ey

For me, the hardest part would be trying to catch up on JavaScript knowledge. Since this is the first JavaScript program I've coded in, I've made a series of mistakes and have to consult online materials to learn stuff like promise, which is the most used method we have in our game. Aside from the language itself, there are tons of dependencies that I have to check out on top of the language, which makes it very time consuming especially if I don't understand the materials.

Adam

The challenges that were faced on my side were generally in relation to the middlewares and debugging. Although I was familiar with javascript I was not familiar with all the

middlewares that we were using, and therefore getting used to the methods, and understanding what they do took some time. In addition, since we needed to have the middlewares work in tandem with one another, there was an increased difficulty in that I needed to understand all the middlewares and how they communicate. In order to better understand all this I relied heavily on debugging in order to try to comprehend where my misunderstandings where. Hence, a lot of time was spent on debugging throughout this project.