Section C - 972 words

Dependencies:

Backend:

- Apollo server express - used for connecting graphql to the client
- Bcrypt - Hash and salt program , used to hash and verify passwords.
- Cookie-parser - Used to parse browser cookies, for storing tokens.
- Cors - Used so the server can work on non local IP addresses.
- Dot env - Used to store secure environment variables.
- Express - Framework for node js to speed up development.
- Jsonwebtoken - Used to create and verify tokens.
- Pg - Allows connections to be made to Postgresql.
  - Alternatively, could use any sql database, MySql is more suited for beginners.
- Type-graphql - Typed GraphQL to select specific instead of all data queried.
- Typeorm - Typed ORM to connect to Database and speed up development.
- Typescript (dev) - Typed version of javascript to make the program more robust, and code more readable.
  - Alternatively, use type definitions in javascript for a less bulky workflow.

Frontend:

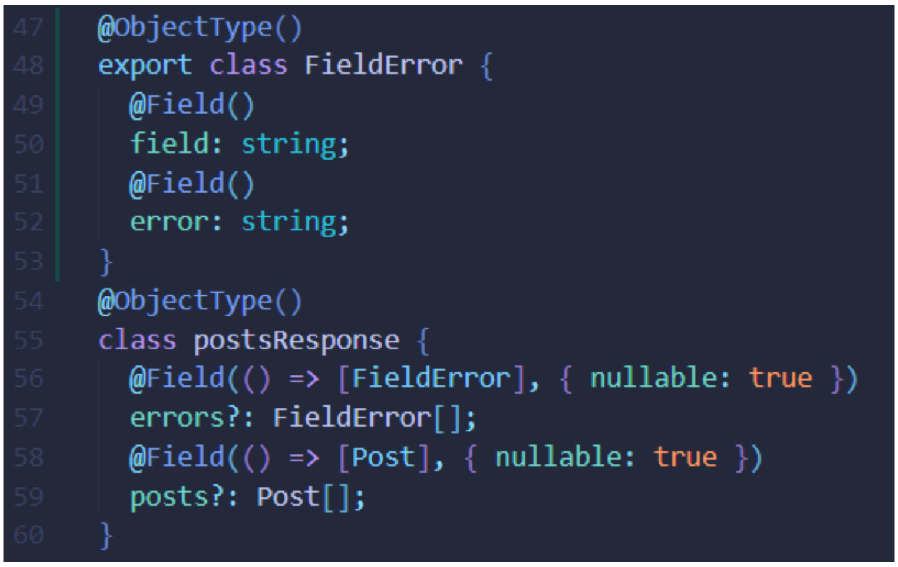
- Apollo client - Used for interacting with graphql API
- GraphQL - Allows selecting specific data needed from query, rather than all.
  - A REST API would meet the criterion better, but allow for faster development.

- React - Framework that allows single components to be rerendered . Development is very slow in javascript without a framework for forums.
    - Alternatively, use vue for a simpler experience.
- React-router - used for creating multiple pages with different urls in React.
- Tailwind - CSS framework to make designing a visually appealing website easier.
    - Bootstrap is the most popular css framework alternative, one of the benefits is that class names are not repeated a lot, like in tailwind.But slightly less customizable.
- Typescript (dev) - Typed version of javascript to make the program more robust, and code more readable.

Typescript, TypeORM and Type-GraphQL

Typescript allows me to strongly type some of my code, which makes it more robust. This is combined with TypeORM and TypeGraphQL for querying the db and server. Typing variables allow me to define variables as having one or more specific types. This is especially useful for validating server responses as it ensures inputs are in the correct type. The reason I used an ORM (TypeOrm) to connect to my database is so that I can create many-to-one relations between my User and post/arguments, which requires creating a table every query. Also it allows me to put my variables in javascript object form, which makes the code more readable.
An example of why this is useful would be the generic server response I give.
I use a graphQL object type decorator and typescript class to define what my graphql server response should look like. This allows me to include a subclass of a

structured error message and the entity I am querying. The (dictionary) FieldError message is useful as it allows me to pin elements to my errors on the client side. The use and application of this technology was mostly influenced by (Awad Ben, 2020), however this tutorial is very outdated code wise and is very simple in comparison to my product.



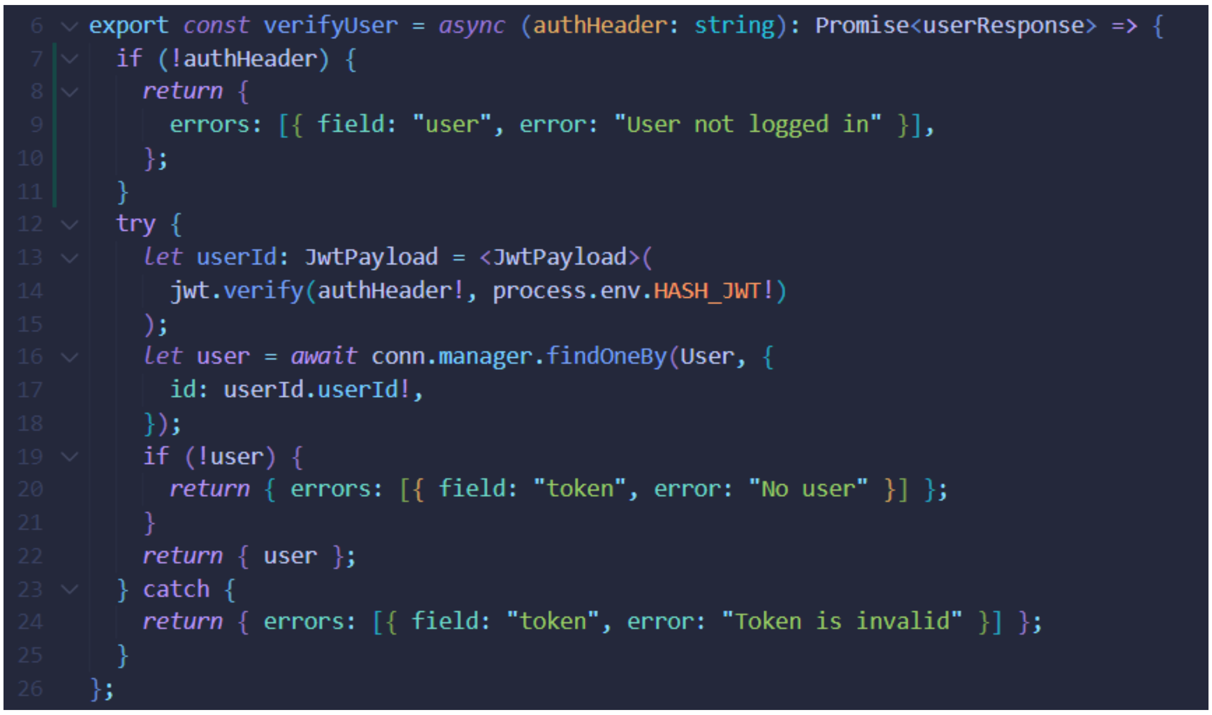
```
@ObjectType()
export class FieldError {
  @Field()
  field: string;
  @Field()
  error: string;
}
@ObjectType()
class postsResponse {
  @Field(() => [FieldError], { nullable: true })
  errors?: FieldError[];
  @Field(() => [Post], { nullable: true })
  posts?: Post[];
}
```

| Example of a response format type (this one is for an array of posts)

User verification

Quite a few functions require the user to be “logged in” (to have a user associated with that action (refer to the UML diagram in Section B). At the beginning I just had the client send a request when loading the page, however this didn’t let me verify the query when testing the database. It could also allow for attacks on the server. Therefore, I changed the server query to a function that I could include at the beginning of other functions.

I also used the return keyword inside of an if statement to avoid using unnecessary if else statements.

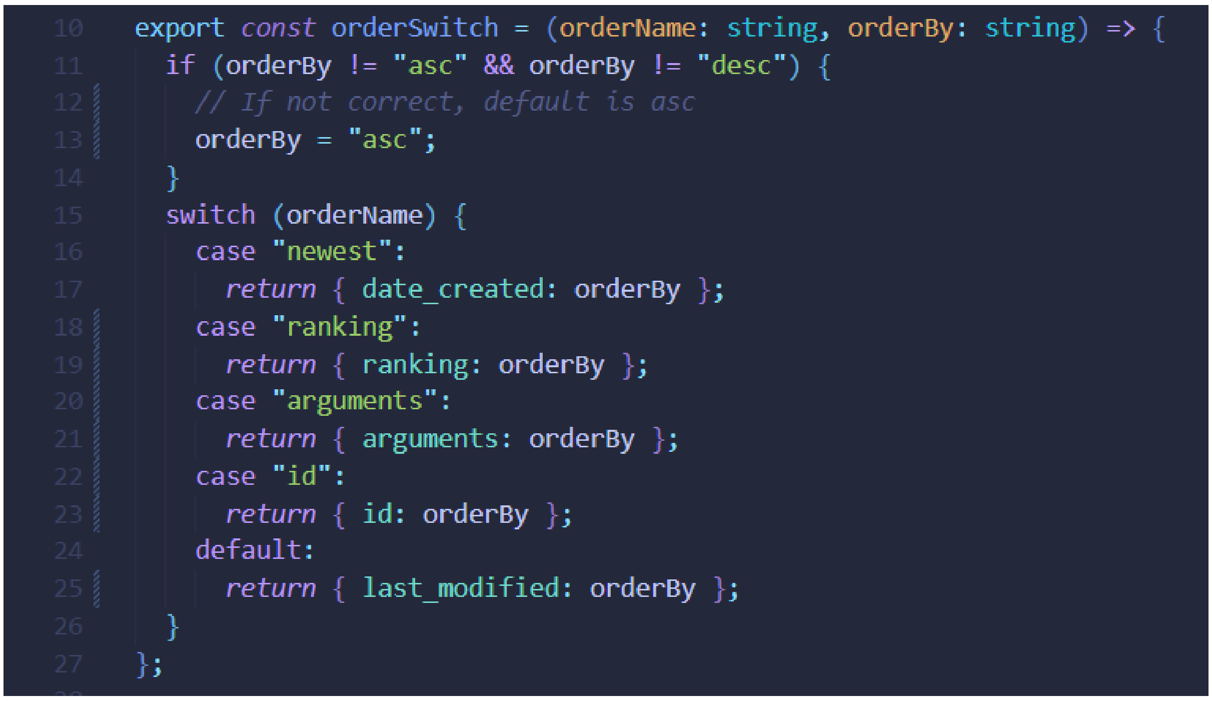


```
export const verifyUser = async (authHeader: string): Promise<userResponse> => {
  if (!authHeader) {
    return {
      errors: [{ field: "user", error: "User not logged in" }],
    };
  }
  try {
    let userId: JwtPayload = <JwtPayload>(
      jwt.verify(authHeader!, process.env.HASH_JWT!)
    );
    let user = await conn.manager.findOneBy(User, {
      id: userId.userId!,
    });
    if (!user) {
      return { errors: [{ field: "token", error: "No user" }] };
    }
    return { user };
  } catch {
    return { errors: [{ field: "token", error: "Token is invalid" }] };
  }
};
```

| Verify user function, method influenced by (Codes Cooper, 2020)

Switch statement

I needed the functionality to order my post queries (the user's feed) in multiple ways. Both for specific use cases on the website, and so that the user can search the website the way they want. I wanted to use a whitelist of words so I couldn't simply turn my string into a dictionary. I chose to use a switch statement instead of an if else, because it is better suited to having a large amount of cases. If this was an if else it would be slower and visually messier/longer.

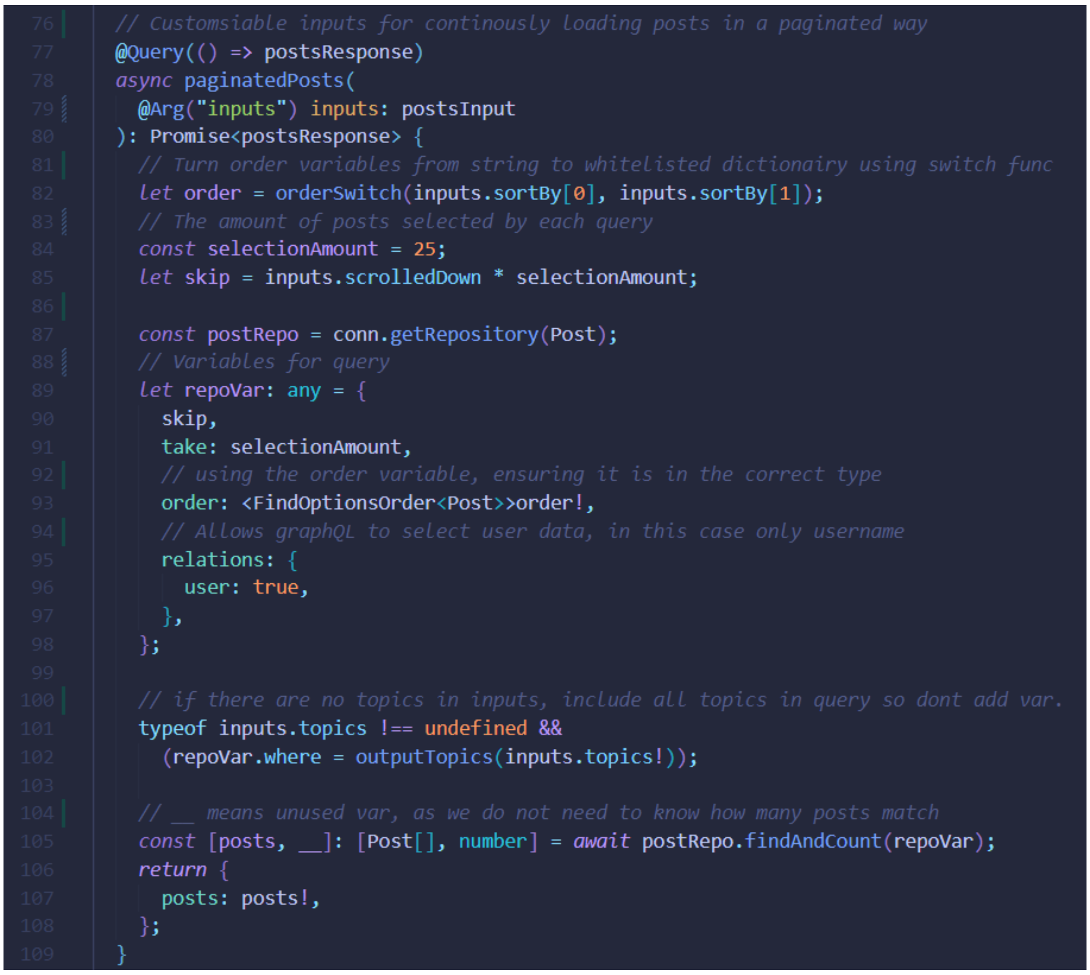


```
export const orderSwitch = (orderName: string, orderBy: string) => {
  if (orderBy != "asc" && orderBy != "desc") {
    // If not correct, default is asc
    orderBy = "asc";
  }
  switch (orderName) {
    case "newest":
      return { date_created: orderBy };
    case "ranking":
      return { ranking: orderBy };
    case "arguments":
      return { arguments: orderBy };
    case "id":
      return { id: orderBy };
    default:
      return { last_modified: orderBy };
  }
};
```

| The switch statement to get the data in the correct form for the database query

Custom querying of posts

Initially, I worked on a pagination system where each query would update after each query. This allowed for the program to limit the amount of data loaded per request and was part of meeting the success criteria. However, I then found I needed to make the queries customisable for certain needs, as well as added user functionality. Therefore I needed to add a dictionary input where I could modify how the data was ordered. As Well as the above switch statement and an inline if condition.



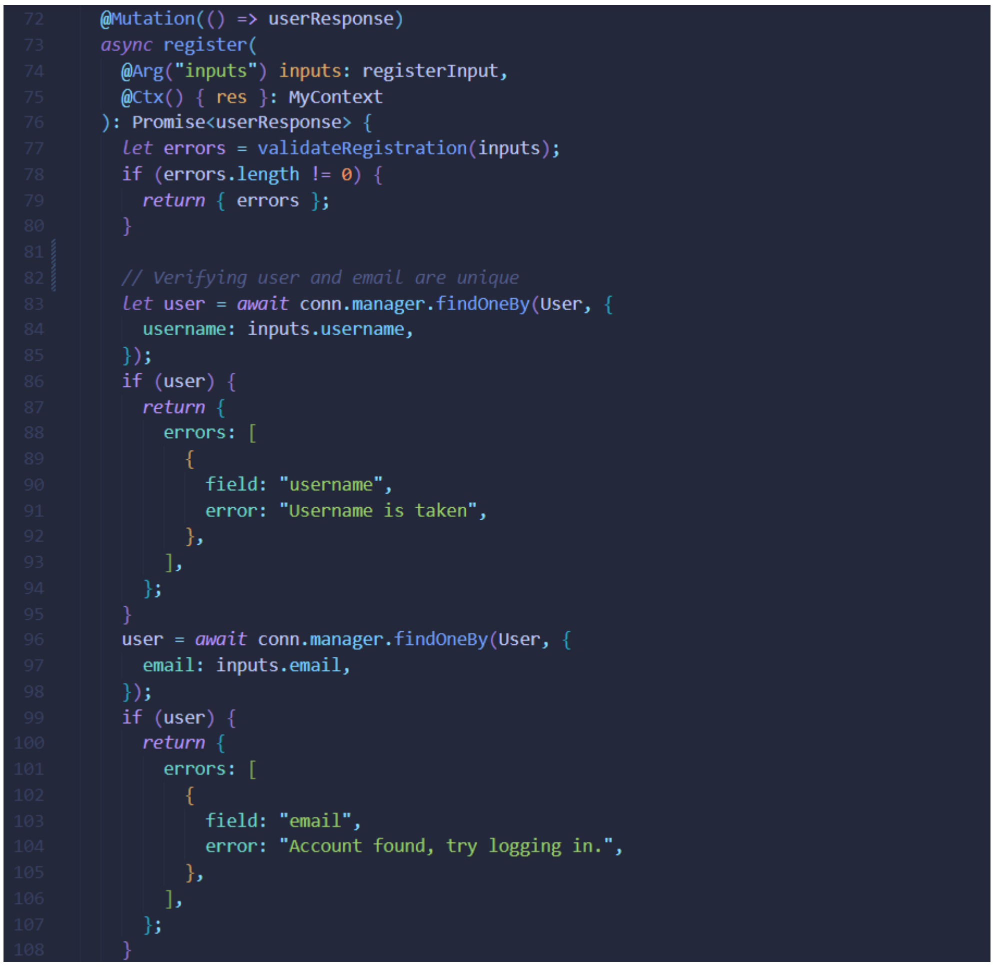
```typescript
// Customsiable inputs for continously loading posts in a paginated way
@Query(() => postsResponse)
async paginatedPosts(
  @Arg("inputs") inputs: postsInput
): Promise<postsResponse> {
  // Turn order variables from string to whitelisted dictionairy using switch func
  let order = orderSwitch(inputs.sortBy[0], inputs.sortBy[1]);
  // The amount of posts selected by each query
  const selectionAmount = 25;
  let skip = inputs.scrolledDown * selectionAmount;

  const postRepo = conn.getRepository(Post);
  // Variables for query
  let repoVar: any = {
    skip,
    take: selectionAmount,
    // using the order variable, ensuring it is in the correct type
    order: <FindOptionsOrder<Post>>order!,
    // Allows graphQL to select user data, in this case only username
    relations: {
      user: true,
    },
  };

  // if there are no topics in inputs, include all topics in query so dont add var.
  typeof inputs.topics !== undefined &&
    (repoVar.where = outputTopics(inputs.topics!));

  // __ means unused var, as we do not need to know how many posts match
  const [posts, __]: [Post[], number] = await postRepo.findAndCount(repoVar);
  return {
    posts: posts!,
  };
}
```

| Code for querying posts in a customisable way

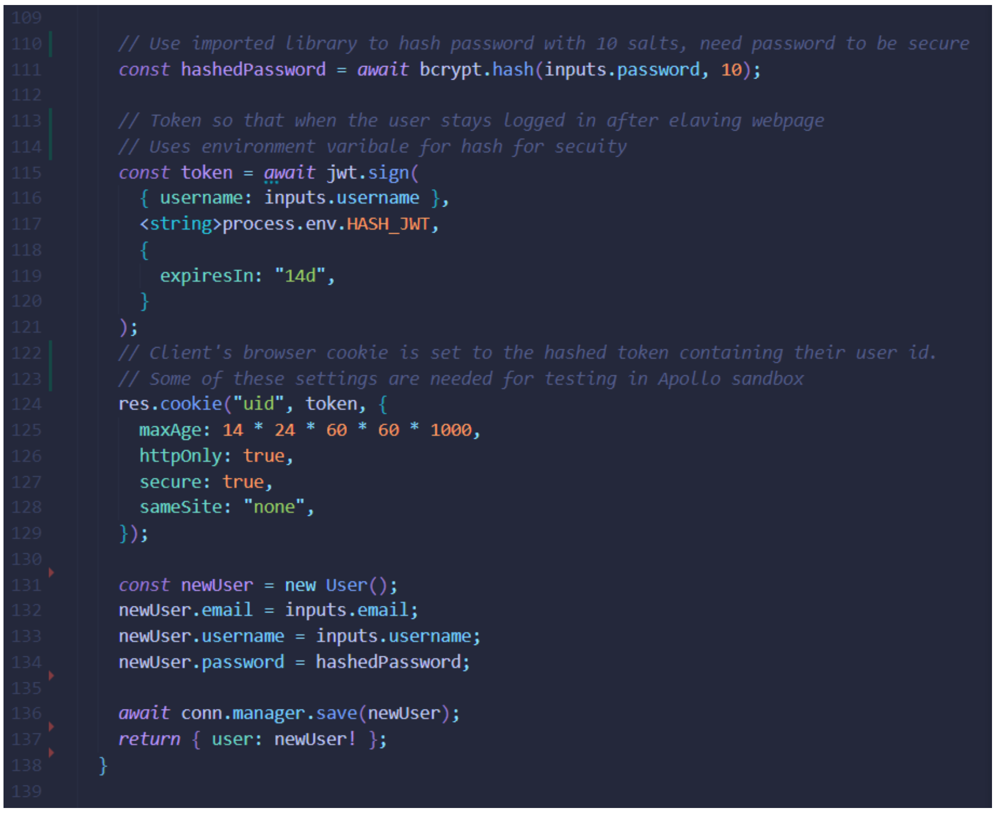
Creating a user

In order to create a user, I needed to:

- validate my form inputs,
- ensure that no users with the unique parameters existed
- Hash and salt my password using bcrypt to ensure it is secure in the database for security reasons.
- Use Jsonwebtoken to create a token of the user id
- Save that token in the client's cookie in the browser
- Save user into Database



```
@Mutation(() => userResponse)
async register(
  @Arg("inputs") inputs: registerInput,
  @Ctx() { res }: MyContext
): Promise<userResponse> {
  let errors = validateRegistration(inputs);
  if (errors.length != 0) {
    return { errors };
  }

  // Verifying user and email are unique
  let user = await conn.manager.findOneBy(User, {
    username: inputs.username,
  });
  if (user) {
    return {
      errors: [
        {
          field: "username",
          error: "Username is taken",
        },
      ],
    };
  }
  user = await conn.manager.findOneBy(User, {
    email: inputs.email,
  });
  if (user) {
    return {
      errors: [
        {
          field: "email",
          error: "Account found, try logging in.",
        },
      ],
    };
  }
```

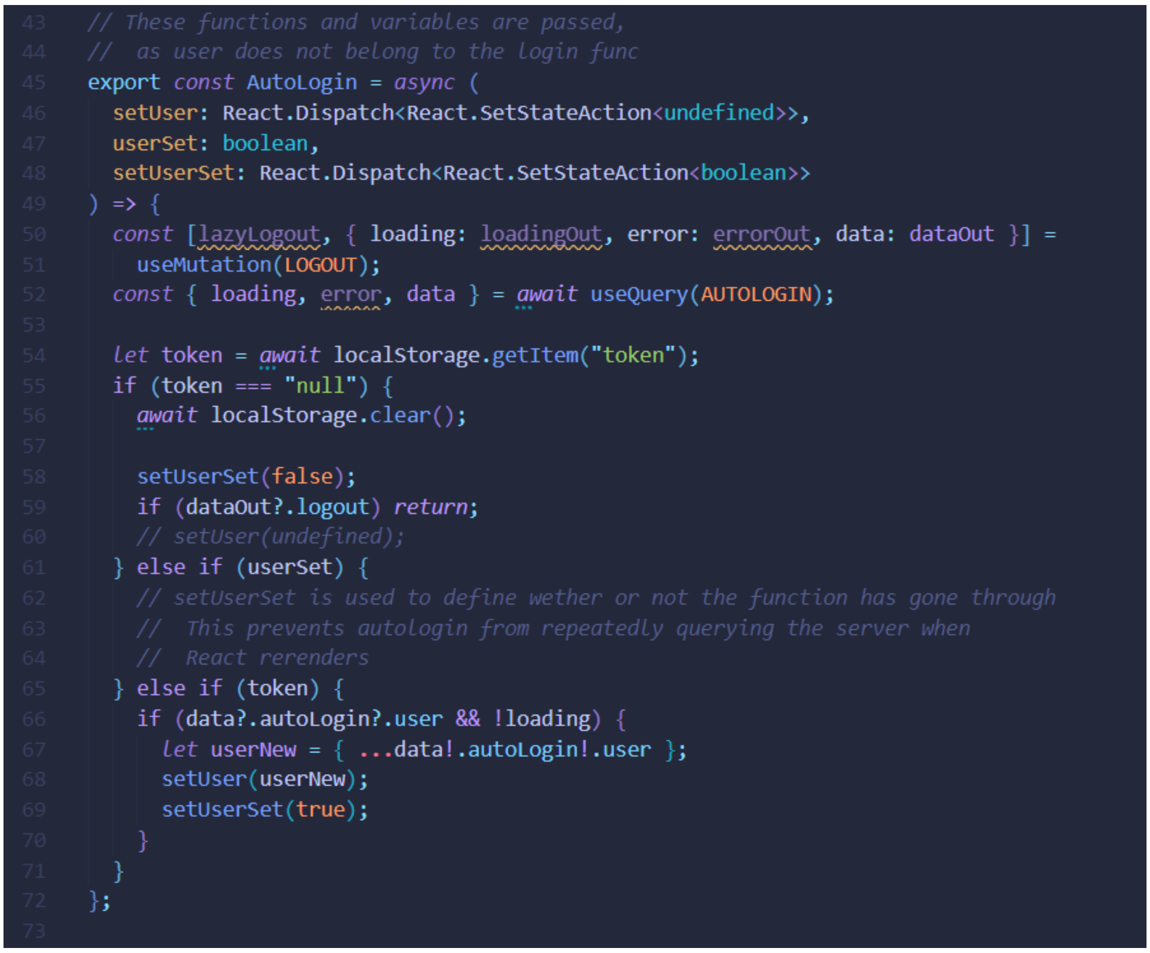


```

    // Use imported library to hash password with 10 salts, need password to be secure
    const hashedPassword = await bcrypt.hash(inputs.password, 10);

    // Token so that when the user stays logged in after elaving webpage
    // Uses environment varibale for hash for secuity
    const token = await jwt.sign(
      { username: inputs.username },
      <string>process.env.HASH_JWT,
      {
        expiresIn: "14d",
      }
    );
    // Client's browser cookie is set to the hashed token containing their user id.
    // Some of these settings are needed for testing in Apollo sandbox
    res.cookie("uid", token, {
      maxAge: 14 * 24 * 60 * 60 * 1000,
      httpOnly: true,
      secure: true,
      sameSite: "none",
    });

    const newUser = new User();
    newUser.email = inputs.email;
    newUser.username = inputs.username;
    newUser.password = hashedPassword;

    await conn.manager.save(newUser);
    return { user: newUser! };
  }

```

| Code snippet for creating the user on the server side influenced by (Codes Cooper, 2020) and (Awad Ben, 2020)

React useState and rerendering (Front end code)

React useState is a variable and the function used to update said variable. It is used for important variables, when these variables are changed, any component that uses them is re rendered to display the updated information. In the autologin function, I use two useStates for one variable. One to check when the variable is set and the other to set the userVariable. This is due to the fact that my application changes the user variable alot (debates/arguments/rankings), so to prevent rerenders and requerying my server for the autologin function, I check whether the setUser variable is true. This is a good example of dealing with the unique problems of React rerenders. This is a functional component, but the styling was heavily influenced by (Themesberg, 2023) and (HeadlessUI, 2023).



```
// These functions and variables are passed,
//  as user does not belong to the login func
export const AutoLogin = async (
  setUser: React.Dispatch<React.SetStateAction<undefined>>,
  userSet: boolean,
  setUserSet: React.Dispatch<React.SetStateAction<boolean>>
) => {
  const [lazyLogout, { loading: loadingOut, error: errorOut, data: dataOut }] =
    useMutation(LOGOUT);
  const { loading, error, data } = await useQuery(AUTOLOGIN);

  let token = await localStorage.getItem("token");
  if (token === "null") {
    await localStorage.clear();

    setUserSet(false);
    if (dataOut?.logout) return;
    // setUser(undefined);
  } else if (userSet) {
    // setUserSet is used to define wether or not the function has gone through
    //  This prevents autologin from repeatedly querying the server when
    //  React rerenders
  } else if (token) {
    if (data?.autoLogin?.user && !loading) {
      let userNew = { ...data!.autoLogin!.user };
      setUser(userNew);
      setUserSet(true);
    }
  }
};
```

| Autologin function

Bibliography

Awad, Ben. “Benawad/Lireddit.” GitHub, 20 Aug. 2020, https://github.com/benawad/lireddit. & https://youtu.be/I6ypD7qv3Z8

Codes, Cooper. “React login with Apollo Server, Context, JWT” 13 Feb 2020, https://youtu.be/0Z68AHS011Y

“Headless UI.” Headlessui.dev, headlessui.com/react/listbox. Accessed 4 Mar. 2023.

“Mutations in Apollo Client.” Apollo Docs, www.apollographql.com/docs/react/data/mutations/. Accessed 3 Mar. 2023.

Themesberg. “Tailwind CSS Forms - Flowbite.” Flowbite.com, flowbite.com/docs/components/forms/. Accessed 3 Mar. 2023.