Online Book Exchange Platform

Sravya Akula

CS628 Full Stack Development II, School of Technology and Computing, MSCS, City University of Seattle

Abstract

Online Book Exchange System is a web platform which is used to exchange books online. It offers an inexpensive way for people to exchange books, find out about new books and obtain a new book to read without having to pay. Instead of selling books back for a fraction of the cost, people can participate in a book exchange by agreeing to exchange their book with someone who requests it and receive the book they requested in return. In this way their old book finds a reader, and in return, the user can get a new collection of books to read which are posted by other users for exchange. Any user can register to search for a book and obtain the owner's information. The user can also put in a request online to exchange a book. The objective of the website being developed is to facilitate an online Book exchange platform which provides quality services and a transparency in book exchanging process. Some of the main functionalities of website are Login & Registration, user account with the personal information, a User can Create, Update, and delete the Book Posts, A search bar where user can search for relevant books. The system is built on top of the MERN stack using GraphQL API Backend. GraphQL is a stateless, client independent API for data exchange with higher query flexibility. MERN is a popular full-stack framework consisting of MongoDB, Express.js, REACT, and NodeJS. MERN stack has made development speedy, reliable, and cost-effective; moreover, it has a complete JavaScript framework that gives the developers the liberty of code sharing. The developers aim to build the system completely written using JavaScript technologies from frontend to backend and at the database level. The back-end server uses Node.js, Express and Mongoose for interacting with MongoDB database. The front-end will be created with React. A simple Express app will be created to perform Create, Read, Update and Delete (CRUD) operations in database with single/multiple collections containing documents for data storage.

1. INTRODUCTION

Book exchange system is developed with a purpose of helping users to Swap books. In this era of e-books not everyone loves e-books, some people love to read physical books rather than e-books. The concept of exchanging books can reduce the carbon footprint. And it encourages reusability of a book. Going to the library or shop for books is time-consuming. Moreover, if we borrow a book from the library, we will have the hassle of renewing it every time. With this online platform people can save money and time by giving an already read collection of books to the ones who need them and get a new collection of books for themselves which they always wanted to have. Book Exchange is a way to make their practice budget-friendly, reusable and to increase the longevity of usage of a book by changing hands. Features of the Online Book Exchange Systems are registration and login of the user, add profile of the user, adding a new book, purchasing a book.

Keywords: Full Stack, MERN Stack, JavaScript, MongoDB, NodeJS, frontend, backend, CRUD, React, GraphQL, Online Exchange.

2. MERN STACK OVERVIEW

MERN Stack is the Full-Stack application that uses JavaScript as the development language. MERN stands for MongoDB, Express, REACT, and Node.js. Each of the initials in MERN stands for a component in the stack:

MongoDB
A database server that is queried using JSON (JavaScript Object Notation) and that stores data structures in a binary JSON format. It organizes the database application and makes conversion among clients and servers easier.

Express
A server-side JavaScript framework to develop single or multi-page web applications.

React
A front-end JavaScript library for building user interfaces based on UI components, permitting the UI to be dynamical.

Node.js
A JavaScript runtime for easily building fast and scalable network applications.

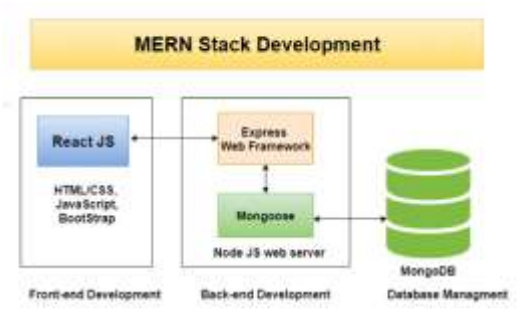


Fig 1: MERN Stack
Source: bocasay.com

Why is MERN Stack chosen for the project?
The major benefit of the MERN stack is that it is extremely quick to prototype with. MERN stack makes the development process speedy, uncomplicated, reliable, and cost-effective. One of the most important benefits of choosing MERN stack for web app development is that Node.js allows users to use JavaScript on the backend as well as the frontend which can save users from having to learn a separate language. Node.js takes care of the server-side coding, and ReactJS is useful for the client-side coding. This also makes it possible for developers to reuse the codes from backend to frontend. Also, if users need to store and share large volumes of data, it is easier to add a field in MongoDB as compared to other databases. In addition, MERN stack has an infinite set of module libraries for Node.js that developers can use at any time. Therefore, there is no need to develop such modules from scratch, and developers can save a lot of time by skipping this process.

REQUIREMENTS SPECIFICATION

Functional requirements

Customers will interact with the system through an easy-to-use top navigation menu:

- "Home": The Home page is the first page that users will see in application. It will serve as a landing page for the application. It will be a simplistic yet aesthetic looking page which will tell the users more about the purpose of the app and what they can expect from it and will have navigational links to the other pages.

- "Register": The Register page is the first page that users see in application. It provides 4 text fields for entering User's name, email, password, and renter the password. In addition, it will have a command button Register that initiates the Register action. If either of the text fields is left blank, it is an error that will be reported to the user.

- "Login": The login page provides 2 text fields for entering Username, password. In addition, it will have a command button Login that initiates the login action. If either of the text fields is left blank, it is an error that will be reported to the user. If both fields are filled in but there is no record of the username, or the password is incorrect that must also be reported to the user. Users that have not yet registered cannot log in. They must first register by clicking on the register command button.

- "Profile": Once the user is authenticated and is logged in the user will be directed to the profile page where the user can update their additional information like address, contact etc. The profile page will have users complete profile information.

- "Add Book": Provides 6 text fields for entering Book title, ISBN, Author, Genre, URL for Book image, and Pickup Address of the Book. Add book page is where user will be adding the Book for exchanging.

- "Books": allows users to see all available Books for purchase. Book can then be purchased using a single button click. In addition, after user clicks Purchase button a model dialogue will be shown to the user to display the order details having Pickup Address of the Book along with action buttons to delete the order or close the window.

Application Requirements

- Node.js backend allows users to use JavaScript as the server-side language.

- Express: A framework for building wed applications on top of Node.js. It simplifies the server creation process.
- MongoDB: A database to store information for the application
- GraphQL is used to expose CRUD operations on data.
- GraphQL splits API requests into Queries and Mutations.
- A Queries that do not change the state of the data and returns a result only.
- Mutations help to modify server-side data.
- For CRUD operations, we use query for read operation and a mutation to create, update, or delete operations.
- Application is developed in React on frontend, which is made up of React Function components, which returns HTML.
- UI is broken down into multiple individual components and merged them all in a parent component which is the final UI.
- React Navigation includes basic navigation from one page to another, going back to the previous page, passing parameters, listening to events, and more.
- Bootstrap and CSS to build a fully responsive and beautiful front end
- GitHub: A central repository to store and version control the code
- Apollo GraphQL Client with React to fetch, cache, and modify application data, which automatically updates UI

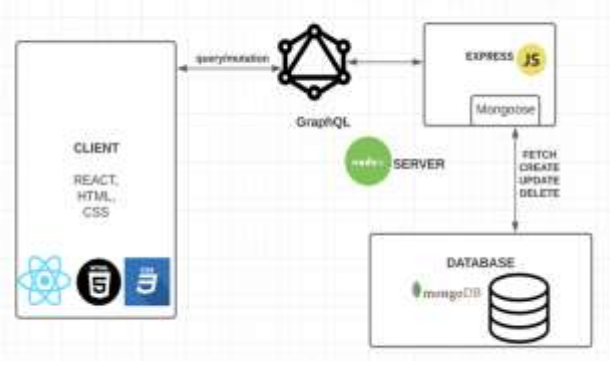


Fig 2: Project components

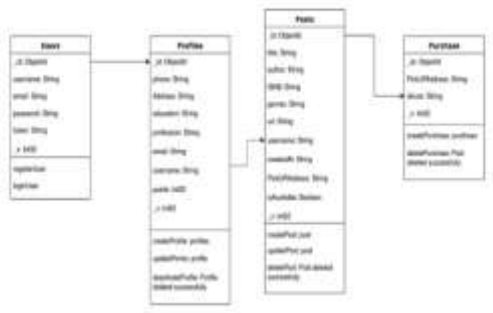


Fig 3: UML diagram of backend tables

Centralized Repository

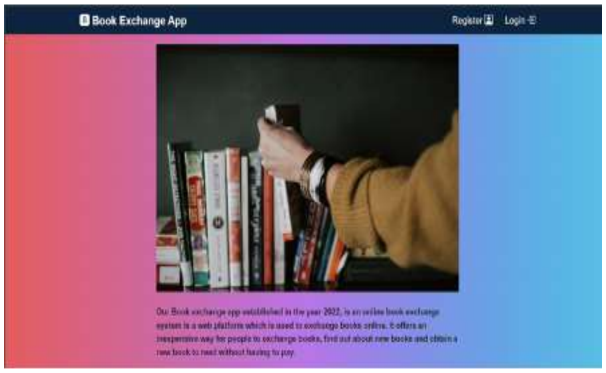
Fig 4: GitHub Repo to store and control the code of Book Exchange System

Database
MongoDB will be used as a database for this project and will contain the collection of documents to store the users, Books, orders data.

Fig 5: MongoDB Atlas Having Collections to Store data

IMPLEMENTATION

Home page:



Our Book exchange app established in the year 2022, is an online book exchange system is a web platform which is used to exchange books online. It offers an inexpensive way for people to exchange books, find out about new books and obtain a new book to read without having to pay.

Figure 6: Homepage

The Home page is the landing page of the application. It is the first page that the users see when they visit the BookExchange app. From this page, they can get a basic idea about the purpose of the app. It gives them an overall idea of the app, contains the logo, the navigation links to allow them to register or if already registered, they can log in through the login page and get authenticated and access the other parts of the application.

Register page:

The register page allows the users to register before they can log into the application and exchange books. There is also a button to navigate to the login page easily, if the user suddenly realizes they had registered previously.

In the registration page we do a few main things:

1. Check for empty fields, valid email formats and password requirements.
2. Once the valid data is inputted by the user, the data is sent to the backend by the GraphQL client.
3. The registerUser mutation is done with the input data.
4. First it is checked if the user already exists or not, if exists, errors caused on backend are displayed to the user.
5. If it is a new user, the new user is created, and the user info is stored in MongoDB with the password stored as encrypted password for security.
6. After getting the created user in the backend, that JWT token is stored in the local Storage. The token is important as it shows that the user has registered successfully.

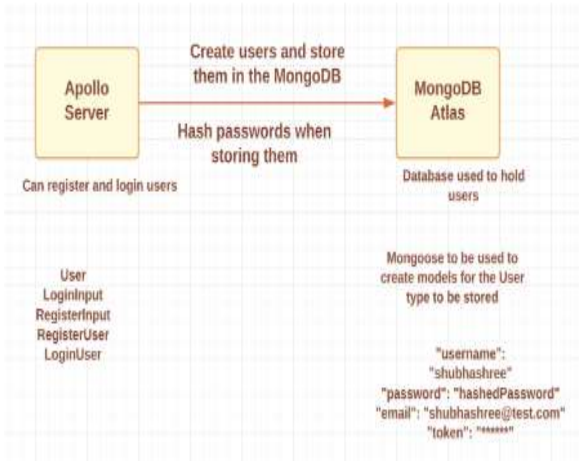

Apollo Server
Create users and store them in the MongoDB
Hash passwords when storing them
MongoDB Atlas
Can register and login users
Database used to hold users
User
LoginInput
RegisterInput
RegisterUser
LoginUser
Mongoose to be used to create models for the User type to be stored
"username": "shubhashree"
"password": "hashedPassword"
"email": "shubhashree@test.com"
"token": "******"

Fig 7: Registration/login Backend

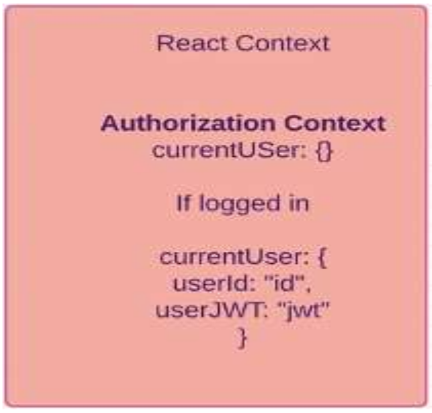


Fig 8: currentUser context

Storing JWT tokens with React:

1. For this React Context is used.
2. Context allows users to store things that the application can interact with anywhere. This means that at any component, page etc., the current User data can be seen.
3. The login/logout functions are created to handle the current User context

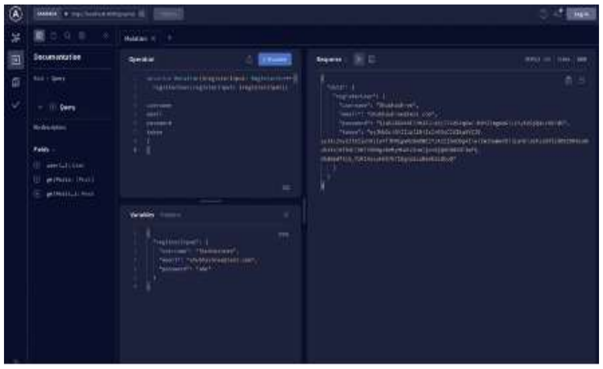


Fig 9: successful registerUser mutation

The screenshot below shows that the user has been successfully registered and an alert is shown to the user that the user has been successfully registered.

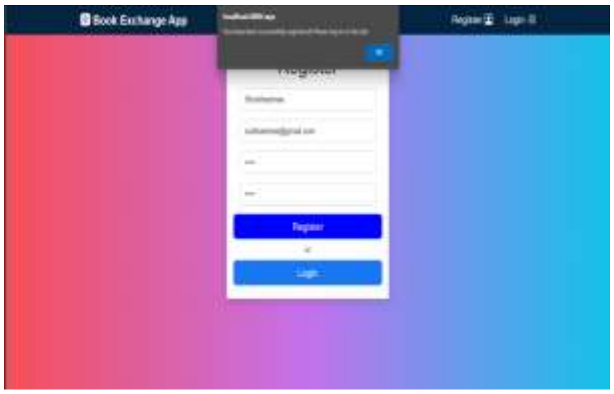


Fig 10: Registration successful

In case, a user already exists in the MongoDB database with the said username, email id, password combination, then the registration will throw an error telling the user that the user already exists in the database and tells the user to go to the login page.

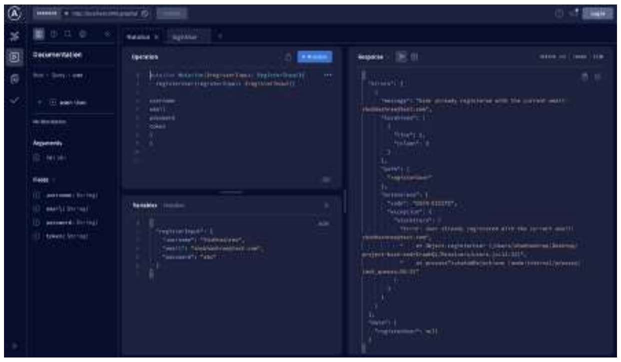


Fig 11: registerUser mutation if the user already exists

In the frontend, the user is also displayed the same error saying that the user already exists in the database. So, in the frontend as well, the notification alert was raised about the existing user.

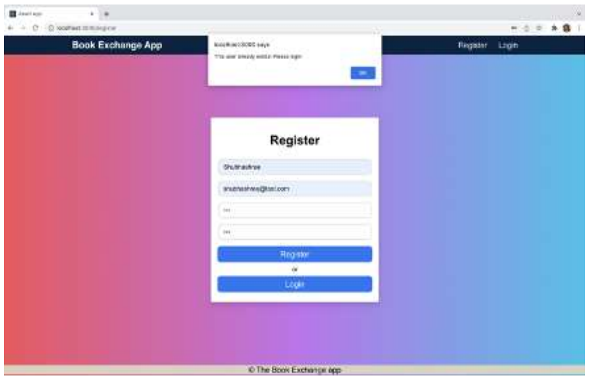


Fig 12: registration page when the user already exists

Login Page:

After completing the registration, the user can log into the app with valid credentials.
On the login page we will do a few main things:

1. The input field values need to be in the correct format.
2. After submitting the login, the user data is checked to see if the user exists in the database.
3. The loginUser mutation is run with the input data.
4. Returns the validated user, or errors caused on backend (e.g., Invalid email/password)
5. After getting successful login, the JWT token is issued for 1 hour and the token is stored in the local Storage.
6. The login/logout functions are created to handle the current User context.
7. After 1 hour, the user is logged out automatically or if the user chooses to log out, there is a logout button and in case the user chooses to log out the token is removed, and the user is set to null.

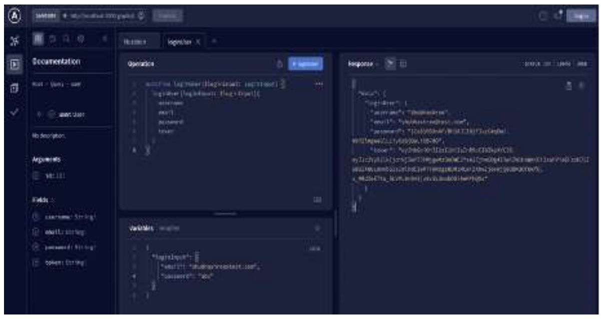


Fig 13: loginUser mutation when login is successful

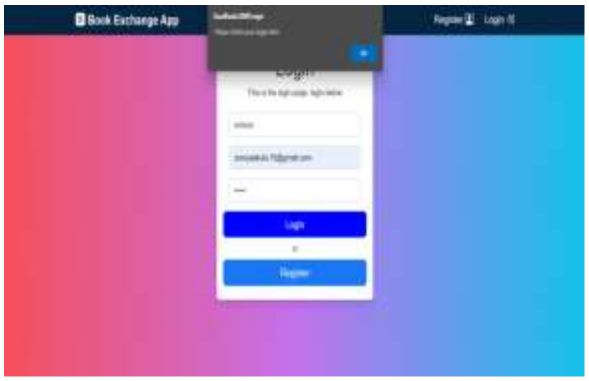


Fig 14: unsuccessful Login with alert

In case of failed login, for instance when the incorrect username, password combination is given, an alert is shown to the user to check the login info and at the backend the GraphQL mutation shows that in case of wrong password, the user can see the password is wrong.

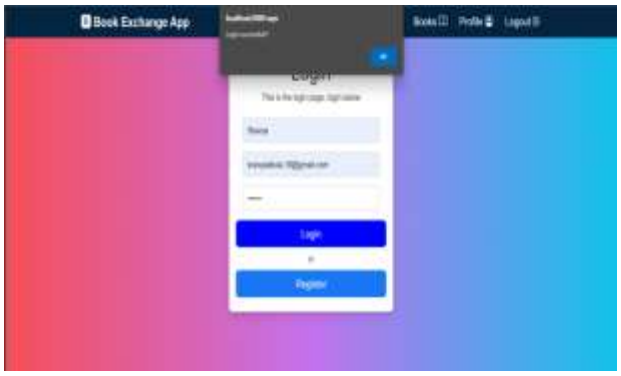


Fig 15: Login successful alert

If the username and password combination is correct, then the user can successfully login and the alert is shown that the login is successful

After successful user login, the user is redirected to the profile page.

Profile page:

Once the user is successfully registered/Logged in, the user will be redirected to profile page where user is allowed to add their additional information like Address and Contact fields to their profile by clicking the manage profile button in the profile page.

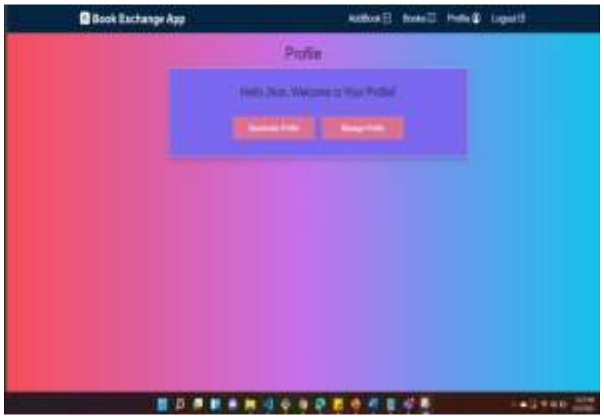


Fig 16: Profile Page

Once the user clicks the manage profile button, the user will be redirected to manage profile page where user can add profile.

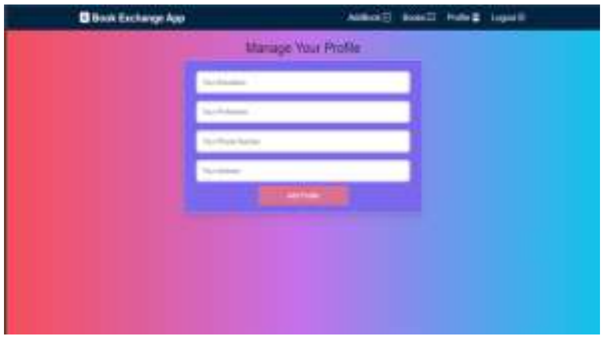


Fig 17: Manage Profile Page

Once the user adds the profile the user will be notified with alert that profile has been added successfully and redirected to the Profile page where user can view the added profile along with the options to manage the profile and delete the profile.

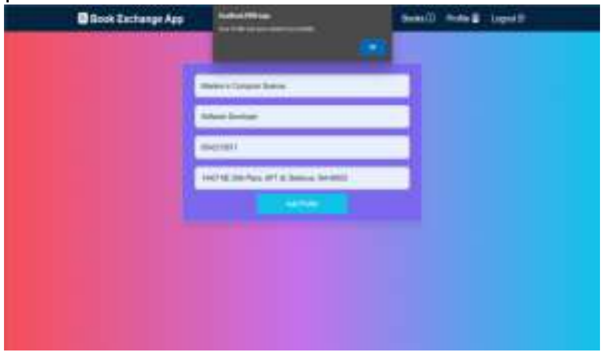


Fig 18: Profile Add Alert

On frontend Usemutation from Apollo GraphQL Client with React is used to create profile which automatically updates UI. When the user adds the fields through interface, user action triggers mutation. The Apollo client issues GraphQL mutation. Then a request is sent to GraphQl API server, which sends the mutation to MongoDB database, where the user profile with details is saved.

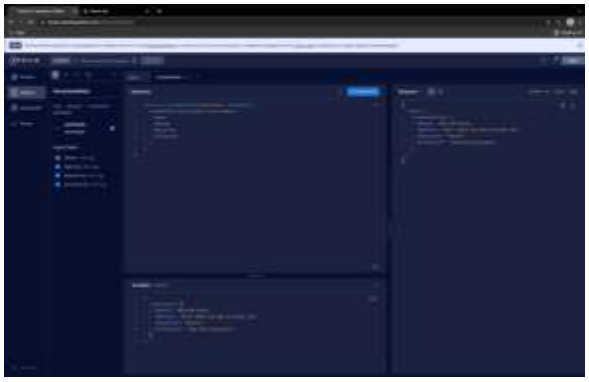


Fig 19: createProfile Mutation

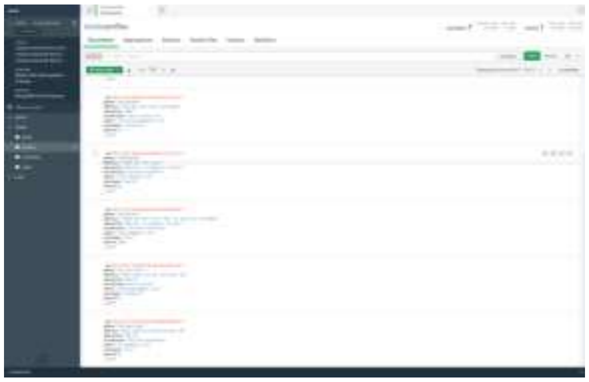
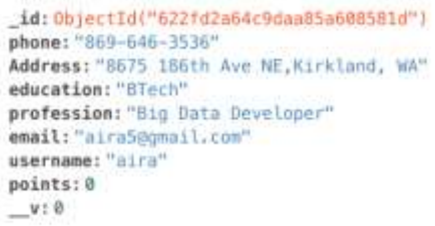


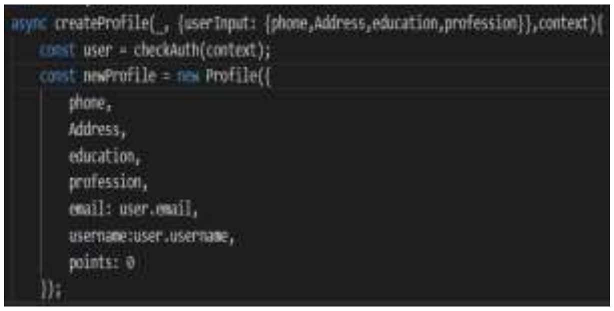
Fig 20: MongoDB Profile Data



```
_id: ObjectId("622fd2a64c9daa85a608581d")
phone: "869-646-3536"
Address: "8675 186th Ave NE,Kirkland, WA"
education: "BTech"
profession: "Big Data Developer"
email: "aira5@gmail.com"
username: "aira"
points: 0
__v: 0
```

Fig 21: Profile data stored in MongoDB

We have added auth middleware check before saving the added user information to the database. Only authorized users will be able to add profile. We fetch usernames from the context. If Unauthorized user try to add a profile it throws a warning message as Invalid User.



```
async createProfile(_, {userInput: {phone,Address,education,profession}},context){
    const user = checkAuth(context);
    const newProfile = new Profile({
        phone,
        Address,
        education,
        profession,
        email: user.email,
        username:user.username,
        points: 0
    });
```

Fig 22: auth middleware check

The Profile page displays the user profile with username, email, phone, Address, education, profession, and points of the user that has been added by the users along with the option to Delete profile and manage profile.

Fig 23: Profile Page with user profile displayed

To get profile that has been added to database, we use GraphQL query, which get added profile fields. When a user adds profile, user action triggers query. The Apollo client issues GraphQL query. Then a request is sent to GraphQl API server, which sends the query to MongoDB database, where the user profile with details is fetched. The graphql server responds to apollo client which changes and drives UI updates in the user interface.

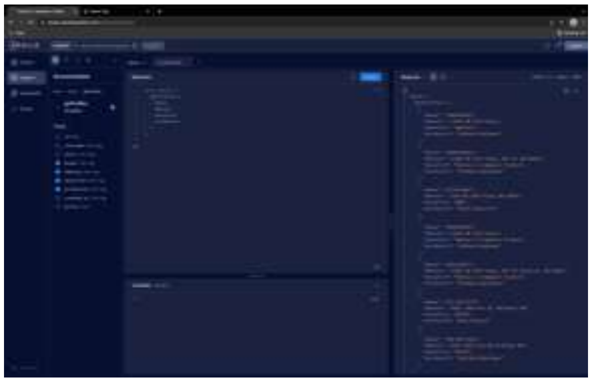
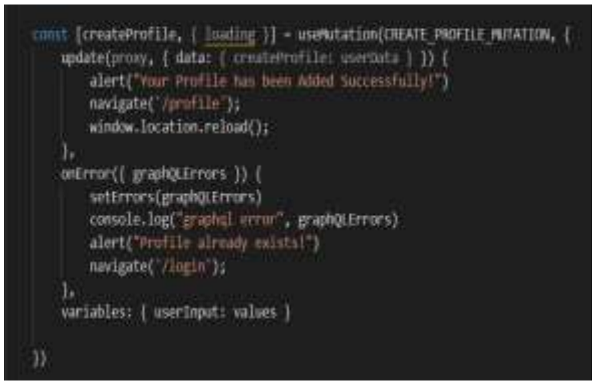


Fig 24: getProfiles Query

The useMutation, useQuery React hook is the primary API for executing mutations in an Apollo application. UseMutation, useQuery returns a tuple that includes a query, mutate function and an object with fields that represent the status of the mutation and query execution (loading, error, and data).



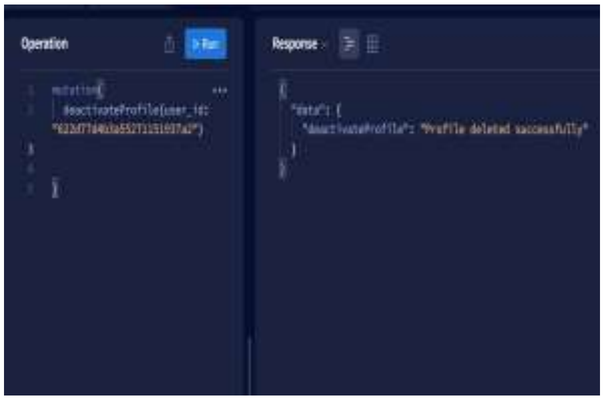
```
const [createProfile, { loading }] = useMutation(CREATE_PROFILE_MUTATION, {
    update(proxy, { data: { createProfile: userData } }) {
        alert("Your Profile has been Added Successfully!")
        navigate('/profile');
        window.location.reload();
    },
    onError({ graphQLErrors }) {
        setErrors(graphQLErrors)
        console.log("graphql error", graphQLErrors)
        alert("Profile already exists!")
        navigate('/login');
    },
    variables: { userInput: values }

})
```

Fig 25: useMutation

Once the user deletes the profile, the user will be notified with an alert saying that the profile has been deleted Successfully.

Fig 26: Profile Delete

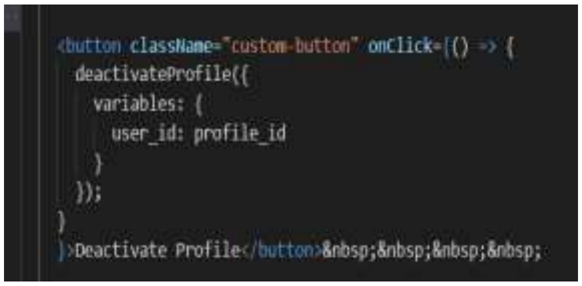
When the user clicks the deactivate profile button, user action triggers mutation. The Apollo client issues GraphQL mutation. Then a request is sent to GraphQl API server, which sends the mutation to MongoDB database, where the user profile is deleted. We pass user id as a parameter to delete mutation to find the user id and delete the user profile of that user id.



```
mutation{
  deactivateProfile(user_id:
"6326f7d40da5527115193fa2")
}
```

```
{
  "data": {
    "deactivateProfile": "Profile deleted successfully"
  }
}
```

Fig 27: deactiveProfile mutation



```
<button className="custom-button" onClick={() => {
  deactivateProfile({
    variables: {
      user_id: profile_id
    }
  });
}
}>Deactivate Profile</button>&nbsp;&nbsp;&nbsp;&nbsp;
```

Fig 28: deactiveProfile mutation frontend

Add Book page:

Add Book is a Page where users can create a Book Post for exchange. The Book Post has Title, Author, ISBN, Genre, URL of the Book Image, Pickup Address of the Book fields.

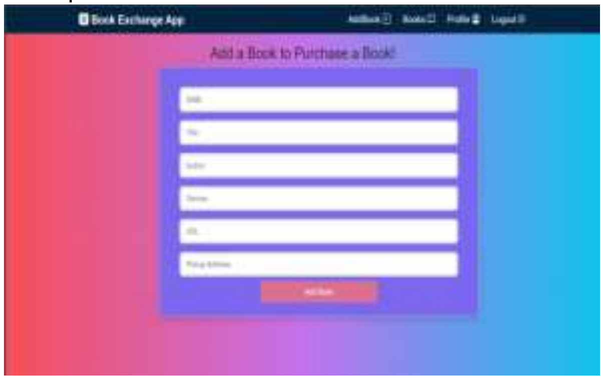


Fig 29: Add Book page

Once the user adds the book post the user will be notified with alert that book has been added successfully and redirected to the Books page where user can select from the available Books.

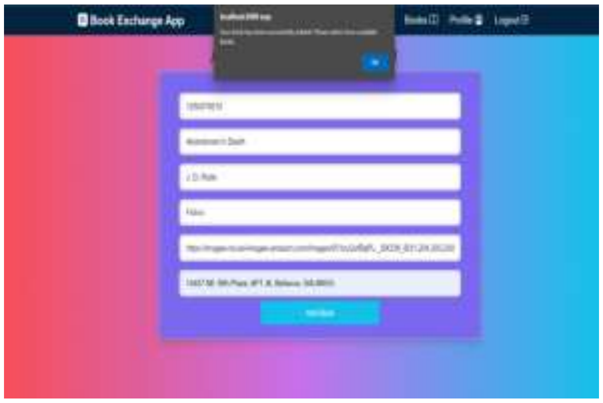


Fig 30: Book successfully added

Once the user posts a Book the user will be credited with points in the Points Filed in the Profile Page with which user can make purchase of books. The number of books the user can purchase depends on the number of books posts the user adds. For one Book Post the user will be credited with 10 points and for one Book purchase the user will be deducted with 10 points.

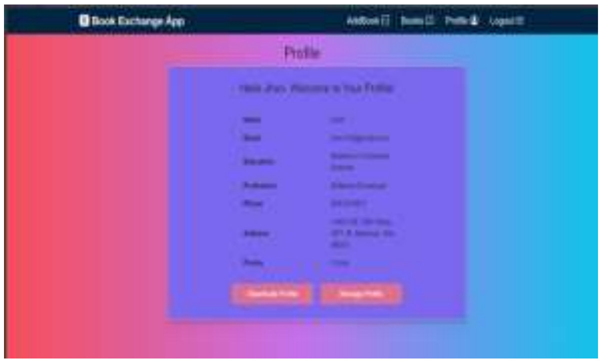


Fig 31: Points Updated in Profile Page

On frontend Usemutation from Apollo GraphQL Client with React is used to fetch, cache, and modify application data, which automatically updates UI. When the user adds the fields through interface, user action triggers mutation. The Apollo client issues GraphQL mutation. Then a request is sent to GraphQl API server, which sends the mutation to MongoDB database, where the Book post with details is saved.

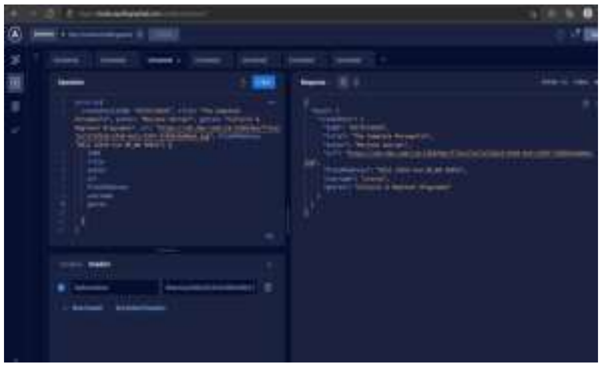
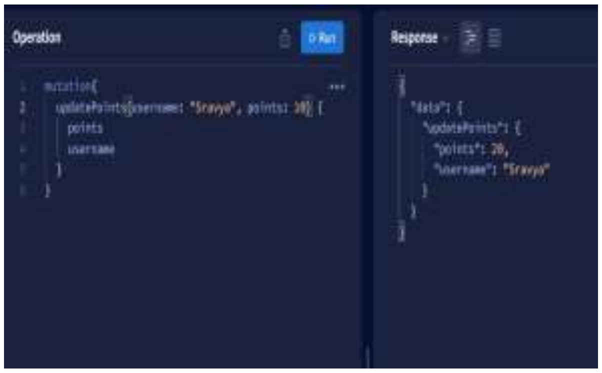


Fig 32: createPost mutation

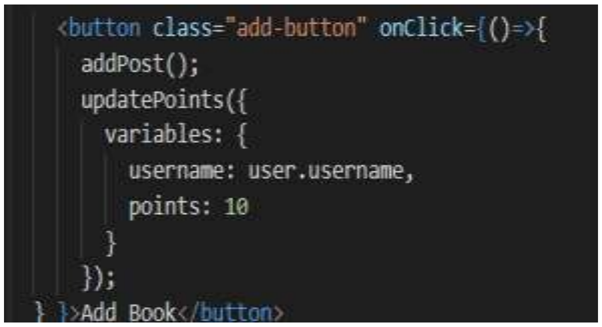
Once user creates a Book Post, we have mutation to update points in the user profile. When the user clicks Add Book through interface, user action triggers mutation. The Apollo client issues GraphQL mutation. Then a request is sent to GraphQl API server, which sends the mutation to MongoDB database, where the points in user profile is updated in the database.



```
mutation{
  updatePoints(username: "Sravya", points: 10) {
    points
    username
  }
}
```

```
{
  "data": {
    "updatePoints": {
      "points": 20,
      "username": "Sravya"
    }
  }
}
```

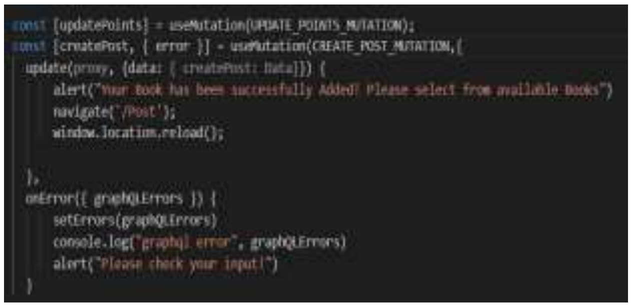
Fig 33: updatePoints mutation



```
  <button class="add-button" onClick={()=>{
    addPost();
    updatePoints({
      variables: {
        username: user.username,
        points: 10
      }
    });
} }>Add Book</button>
```

Fig 34: updatePoints mutation frontend

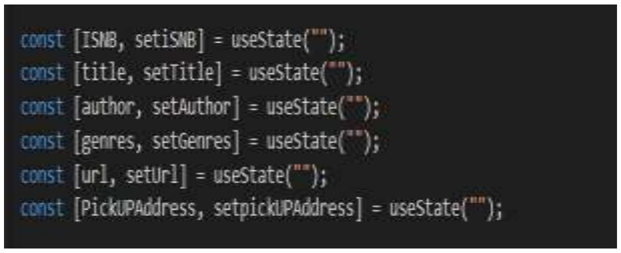
The useMutation React hook is the primary API for executing mutations in an Apollo application. To run a mutation, we first call useMutation within a React component and pass it a GraphQL string that represents the mutation. When component renders, useMutation returns a tuple that includes a mutate function and an object with fields that represent the status of the mutation's execution (loading, error, and data).



```
const [updatePoints] = useMutation(UPDATE_POINTS_MUTATION);
const [createPost, { error }] = useMutation(CREATE_POST_MUTATION,{
  update(proxy, {data: { createPost: Data}}) {
      alert("Your Book has been successfully Added! Please select from available Books")
      navigate('/Post');
      window.location.reload();

  },
  onError({ graphQLErrors }) {
      setErrors(graphQLErrors)
      console.log("graphql error", graphQLErrors)
      alert("Please check your input!")
  }
```

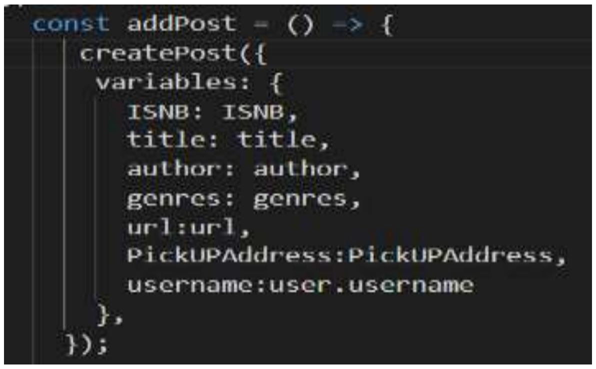
Fig 35: useMutation react

We useState Hook to have state of the all the variables (Title, Author, ISBN, Genre, URL of the Book Image, Pickup Address) in functional components. we pass the initial state to this function, and it returns a variable with the current state value. We have a function addPost to update these variable values.



```
const [ISNB, setiSNB] = useState("");
const [title, setTitle] = useState("");
const [author, setAuthor] = useState("");
const [genres, setGenres] = useState("");
const [url, setUrl] = useState("");
const [PickUPAddress, setpickUPAddress] = useState("");
```

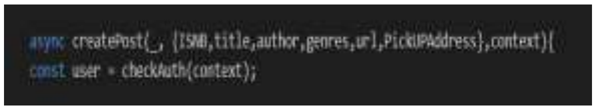
Fig 36: useState Hook to have state of variables



```
const addPost = () => {
    createPost({
      variables: {
        ISNB: ISNB,
        title: title,
        author: author,
        genres: genres,
        url:url,
        PickUPAddress:PickUPAddress,
        username:user.username
      },
    });
```

Fig 37: Function addPost to update variables

We add auth middleware check before the added book by user is saved to database. Only authorized users will be able to add the book. We fetch usernames from the context.



```
async createPost(_, {ISNB,title,author,genres,url,PickUPAddress},context){
const user = checkAuth(context);
```

Fig 38: auth middleware check

The Validation check is added to the Book fields (Title, Author, ISBN, Genre, URL of the Book Image, Pickup Address). If a user tries to add Book Post with empty fields for either of the fields an alert is shown to the user to check the input.

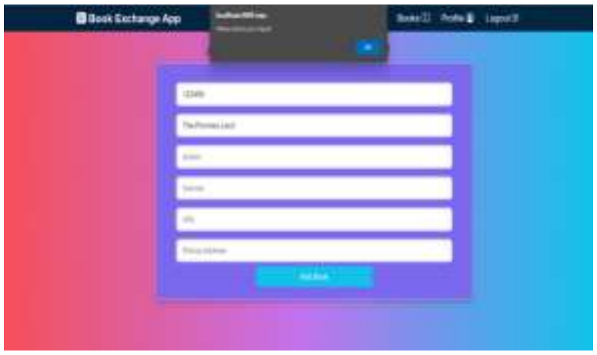


Fig 39 Validation check error

Books Page:

The Books page displays the Book Posts with Title, Author, ISBN, Genre, URL of the Book Image, Username of the Book that have been added by the users along with the option to purchase.

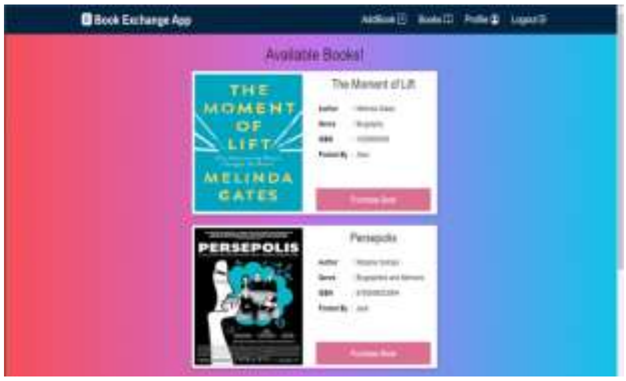


Fig 40: Books Page

The user can purchase a book only if the user has enough points in his profile. Once the user clicks the Purchase Book button a model box with order confirmation message and pickup address of the book along with option to cancel the order are displayed. If the User does not cancel the order, then an order is created for the user and the user points will be deducted.

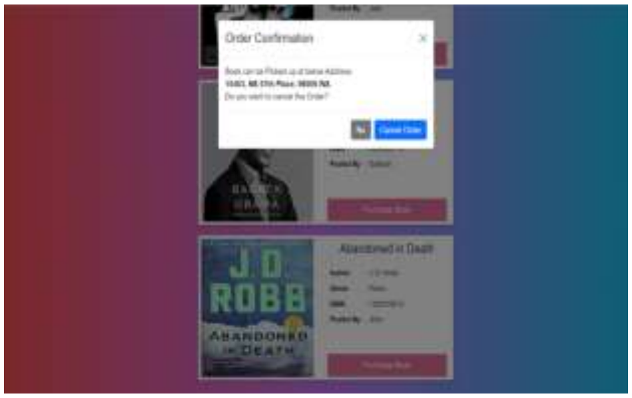


Fig 41: Order Confirmation Model Box

If the user cancels the order the order created will be deleted and user points will be credited back to the user and an alert saying that the Order has been cancelled is displayed.

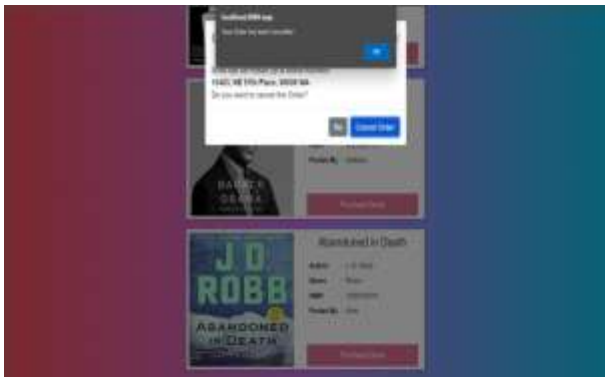


Fig 42: Order Cancel Alert

For A user to make purchase the user must create the profile and should also have enough points to make purchase. If a user tries to make purchase without creating profile an alert saying user to create profile is displayed. If a user tries to make purchase without having enough points in profile account an alert saying that user does not have enough points to order the book is displayed.

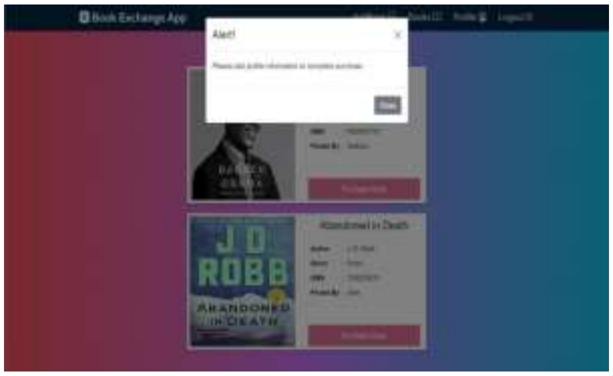


Fig 43: Profile Alert

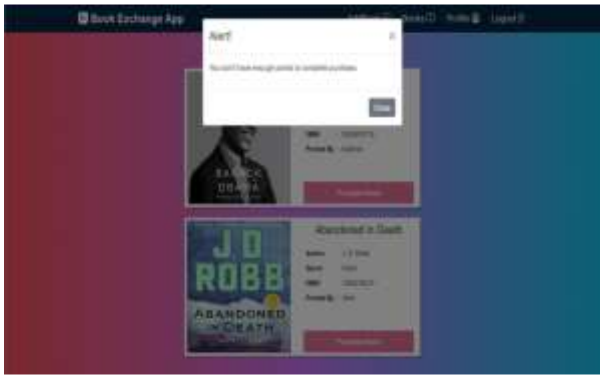


Fig 44: Points Alert

To get books that have been added to database, we use GraphQL query, which gets added Book fields. When a user adds book posts, user action triggers query. The Apollo client issues GraphQL query. Then a request is sent to GraphQl API server, which sends the query to MongoDB database, where the Book post with details is fetched. The graphql server responds to apollo client which changes and drives UI updates in the user interface.

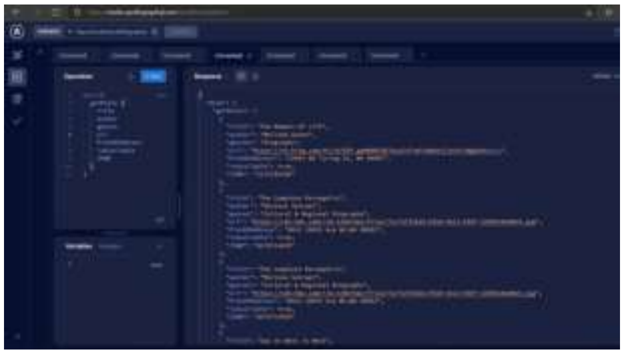
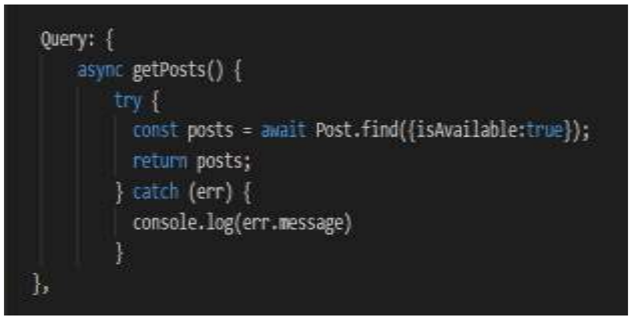


Fig 45: Get Books Query

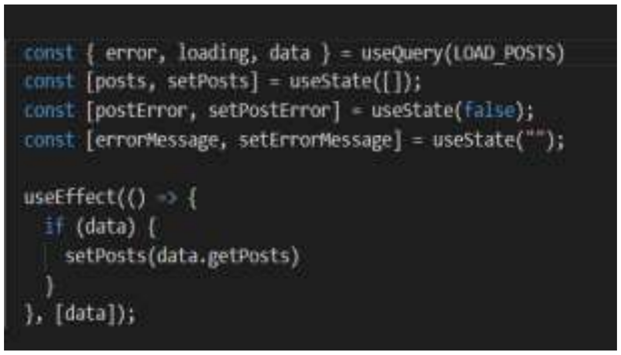
Before displaying the Books, we check if book is available, i.e., already purchased by any user. From the displayed Books the users will be able to view the title, author, genre, image, ISBN, pickup address of the book.



```
Query: {
    async getPosts() {
        try {
          const posts = await Post.find({isAvailable:true});
          return posts;
        } catch (err) {
          console.log(err.message)
        }
},
```

Fig 46: Book Availability Check

UseQuery React is the primary API for executing queries in an Apollo application. To run a query within a React component, we call useQuery and pass it a GraphQL query string. When component renders, useQuery returns an object from Apollo Client that contains loading, error, and data properties that can be used to render UI. We use useState Hook to have state of the Book Posts in functional components. By using useEffect Hook, we tell React that component needs to getPosts after render. React will remembers the function passed (refer to it as "effect"), and call it later after performing the DOM updates



```
const { error, loading, data } = useQuery(LOAD_POSTS)
const [posts, setPosts] = useState([]);
const [postError, setPostError] = useState(false);
const [errorMessage, setErrorMessage] = useState("");

useEffect(() => {
  if (data) {
    setPosts(data.getPosts)
  }
}, [data]);
```

Fig 47: UseQuery React

Once the user clicks Purchase Book Button, the order is created. On frontend Usemutation from Apollo GraphQL Client with React is used to create the order. When the user clicks Purchase Book Button, user action triggers mutation. The Apollo client issues GraphQL mutation. Then a request is sent to GraphQl API server, which sends the mutation to MongoDB database, where the order with details is saved.

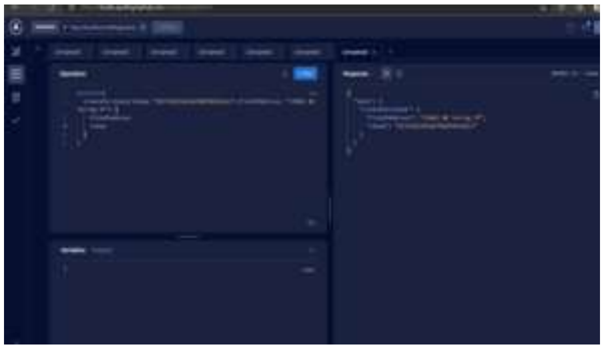
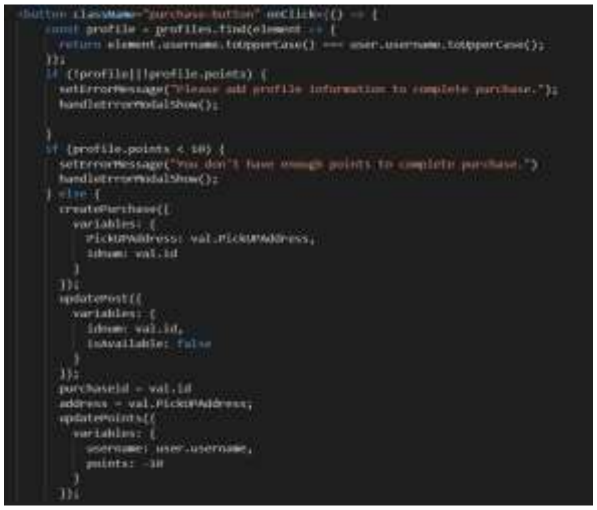


Fig 48: createPurchase mutation

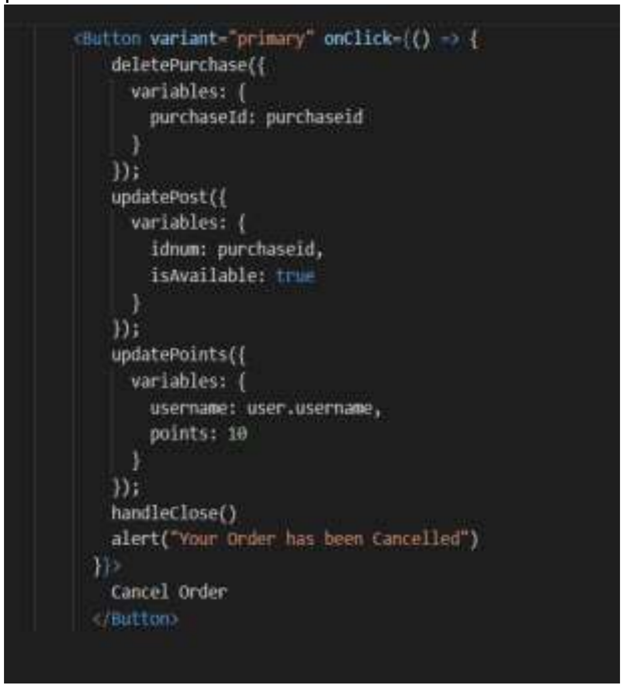
Before creating the order, we check for user Points and Profile. If the user does not have either of these we will be notified by an alert. So, after creating the order we deduct the points in user profile account using update Points mutation and update the Book Post to Not available using updatePost mutation. when user click Purchase button user action triggers all the three mutations (createPurchase, updatePoints, updatePost). The Apollo client issues GraphQL mutations. Then a request is sent to GraphQl API server, which sends the mutations to MongoDB database, where the order is created and posted is updated to not available and update points in the user profile.



```
<Button className="purchase-button" onClick={() => {
    const profile = profiles.find(element => {
      return element.username.toUpperCase() === user.username.toUpperCase();
    });
    if (!profile||!profile.points) {
      setErrorMessage("Please add profile information to complete purchase.");
      handleErrorModalShow();

    }
    if (profile.points < 10) {
      setErrorMessage("You don't have enough points to complete purchase.")
      handleErrorModalShow();
    } else {
      createPurchase({
        variables: {
          PickUPAddress: val.PickUPAddress,
          idnum: val.id
        }
      });
      updatePost({
        variables: {
          idnum: val.id,
          isAvailable: false
        }
      });
      purchaseid = val.id
      address = val.PickUPAddress;
      updatePoints({
        variables: {
          username: user.username,
          points: -10
        }
      });
```

Fig 49: CreatePurchase, updatePoints, updatePost Mutations on frontend

When the user clicks Cancel Order button user action triggers all the three mutations (deletePurchase, updatePoints, updatePost). The Apollo client issues GraphQL mutations. Then a request is sent to GraphQl API server, which sends the mutations to MongoDB database, where the order is deleted and posted is updated to available and update points in the user profile.



```
<Button variant="primary" onClick={() => {
  deletePurchase({
    variables: {
      purchaseId: purchaseid
    }
  });
  updatePost({
    variables: {
      idnum: purchaseid,
      isAvailable: true
    }
  });
  updatePoints({
    variables: {
      username: user.username,
      points: 10
    }
  });
  handleClose()
  alert("Your Order has been Cancelled")
}}>
  Cancel Order
</Button>
```

Fig 50: CreatePurchase, updatePoints, updatePost Mutations on frontend

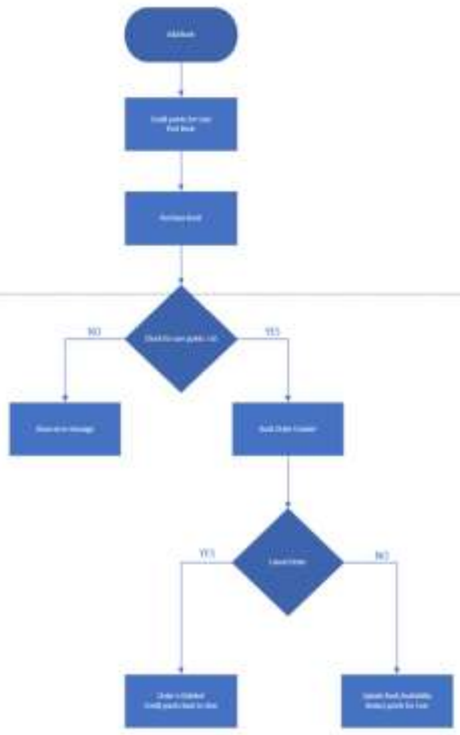


Fig 51: Add Book and Books pages workflow representation.

6. CONCLUSIONS

The Homepage is the landing page of the application where users learn more about the application and can decide whether they want to exchange books or not. In case, they choose to, they must be authenticated before they can exchange books. For that reason, registration and login pages are provided for them to register log in. Unauthenticated users can only access the homepage. In future implementations, single sign-on options like Auth0, OAuth will also be provided to the users to give them a choice to use our login or use the single sign-on option. More security features will also be added to our application. Once the user is authenticated and is logged in the user will be directed to the profile page where the user can update their additional information like address, contact etc. The profile page will have users complete profile information. Add book page is where user will be adding the Book for exchanging. Books page allows users to see all available Books, order a Book, cancel a Book order. A user can purchase a book only if the user has points in his profile account which he can get by adding a book in Add book page.

7. ACKNOWLEDGEMENTS

I would like to thank our instructor, Mr. YuChe Liu, for his guidance throughout the project and the valuable MERN concepts taught by him which have helped the team navigate through the hurdles and learn new concepts while working on the project.

8. REFERENCES

- *Subramanian, V. (2019). Pro MERN Stack: Full Stack Web App Development with Mongo, Express, React, and Node. Express. Berkeley, CA.*

- *Hoque. (2020). Full stacks react projects: learn MERN stack development by building modern web apps using MongoDB, Express, React, and Node.js. (Second edition.). Packt Publishing.*

- *Apollo GraphQL (2021), https://www.apollographql.com/docs/*

- *MongoDB (2021), https://www.mongodb.com/atlas/database*

- *React (2021), https://reactjs.org/docs/components-and-props.html*