Unit Testing

In this assignment, you will modify certain data-fetching endpoints in your Express application and replace them with their equivalents in GraphQL. Additionally, you will update these endpoints in your React application with a GraphQL client.

Initial Setup

- We recommend you create a new branch so that your previous implementation is safely kept for both of your frontend and backend apps.
- In your express application, make sure to install the necessary libraries, graphql and express-graphql.

GraphQL in the backend

Queries and Types Schema Definitions

- Create a GraphQL schema.
- Translate your Mongoose models (`Coder`, `Manager`, `Challenge`, and `Submission`) into graphql types.

For example, the manager model will be translated into the following type:

```
Unset
  type Manager {

      _id: ID!

      first_name: String!

      last_name: String!

      email: String!
```

```
    password: String!
}
```

- Create queries to get all the challenges (They should accept the `category` filter to get challenges of that category only), get a challenge by id and get all the categories.
- Create resolver functions for each query. For the resolvers, you can make use of your previously developed services.

  Note: In our previous implementation, we extracted the authentication token from the header, parsed it, and passed the id and role of the user to the services to get tailored responses (Include the status and solution rate of the challenge for a specific coder for example).

- For this implementation, at first, you can force a token to be passed as an argument to the query, parse it in the resolver, and use it. Once you start integrating GraphQL within React, you are going to provide the token as a request header and retrieve it in the resolver without forcing the token to be present in the query.

- Use `graphiql` interface to check that all endpoints work.

GraphQL in the frontend

Now, you are going to change some react endpoints to use GraphQL. There's a nice integration of graphql and rtk-query using @rtk-query/graphql-request-base-query package and graphql-request used to make graphql requests.

- Create a new API responsible for making GraphQL requests. Here's an example of such an API:

```JavaScript
export const gqlApi = createApi({
  reducerPath: "api",
  baseQuery: graphqlRequestBaseQuery({
    url: "http://localhost:3000/graphql/",
    prepareHeaders: (headers, { getState }) => {
      // retrieve token from redux store
      const token = getState().auth?.token;
      if (token) {
        headers.set("Authorization", `Bearer ${token}`);
      }
      return headers;
    },
  }),
  endpoints: (builder) => ({}),
});
```

As you can see, instead of using the `fetchBaseQuery`, we use `graphqlRequestBaseQuery`. The `prepareHeaders` part is used to get the token from the redux store and inject it into the header so that it can be used by the server for authentication and authorization.

- Make sure to add the API to the redux store as we did previously.
- Create three endpoints:

- Get all categories endpoints that should return the list of categories in the backend.
- Get all challenges, where you should only retrieve the used fields in the table (status, title, category, difficulty level, and solution rate).
- Make sure to include the category filter in the get all challenges endpoint.
- Get a challenge by id that will be used to initialize the coding workspace, so make sure to grab all the data (description, code, submission if there is one, the tests, etc.)
- Make sure to export the created API hooks.
- Replace the three REST rtk-query hooks (get all categories, get all challenges, and get a challenge by id) of your previous with these graphql query hooks.
- Make sure the result remains the same. We just did some sort of optimization by replacing some data-fetching endpoints that caused some over-fetching issues.