Ryan O'Connor

SWE 6853

Term Project

Summer 2023

Table of contents:

Description:

For my project I chose to develop a Distributed System which I named "Quizzard". This is a full-stack web-application system which uses the MERN technology stack. MERN stands for MongoDB(database), Express.js(backend), React.js(frontend), and Node.js(backend). I chose to work with these tools because they are in-demand in the job market.

How the app is intended to work: Admin Users interact with the frontend to create quizzes. Users that are not admins can take quizzes and get a score. When creating a quiz, Admin Users enter a title and add questions along with options. The questions and options are sent to the backend via API requests, which are stored in the MongoDB database (NoSQL). All Users can view existing quizzes and take quizzes created by Admin User.

Functional Requirements:

1. Users should be able to register and log in to the application.

2. Authenticated users can create quizzes.

3. Each quiz can have multiple questions, and each question can have multiple options.

4. Users can view and attempt quizzes.

5. The application should provide feedback on quiz results.

6. Users with appropriate permissions (admins) can manage quizzes, questions, and options.

Entities

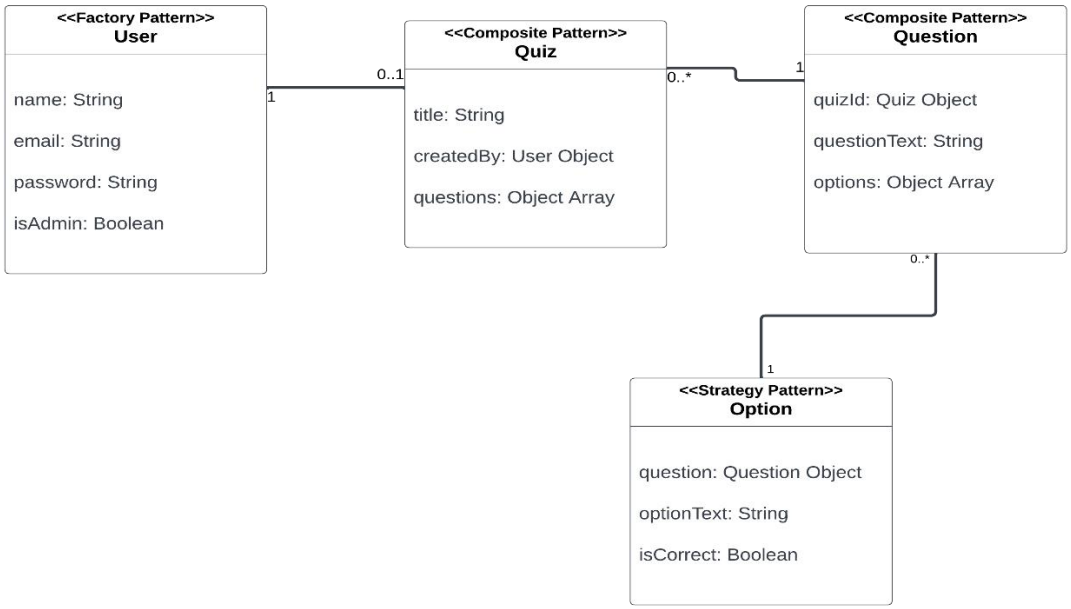
I developed this application around 4 primary entities being: User, Quiz, Question, and Option.

Users have attributes: name, email, password, and isAdmin (for admin privileges).

Quizzes have attributes: title, createdBy (reference to User), and questions (array of references to Question).

Questions have the attributes: quiz (reference to Quiz), questionText, and options (array of references to Option).

Options have the attributes: question (reference to Question), optionText, and isCorrect.



Design Patterns

1. **Factory Pattern:** Used for creating instances of User objects.

```
JS user.js ×
backend > models > JS user.js > ...
//Factory Pattern
const mongoose = require('mongoose');
const Schema = mongoose.Schema;

const UserSchema = new Schema({
    name: { type: String, required: true },
    email: { type: String, required: true, unique: true },
    password: { type: String, required: true },
    isAdmin: { type: Boolean, default: false },
});

module.exports = mongoose.model('User', UserSchema);
```

2. **Composite Pattern:** Used for modeling the hierarchical relationship between Quiz and Question entities.

```
JS quiz.js ×
backend > models > JS quiz.js > ...
//Composite Pattern
const mongoose = require('mongoose');
const Schema = mongoose.Schema;

const QuizSchema = new Schema({
    title: { type: String, required: true },
    questions: [{ type: Schema.Types.ObjectId, ref: 'Question' }],
    createdBy: { type: Schema.Types.ObjectId, ref: 'User' },
});

module.exports = mongoose.model('Quiz', QuizSchema);
```

```
question.js
backend > models > question.js > ...
//Composite Pattern
const mongoose = require('mongoose');
const Schema = mongoose.Schema;

const QuestionSchema = new Schema({
    quiz: { type: Schema.Types.ObjectId, ref: 'Quiz' },
    questionText: { type: String, required: true },
    options: [{ type: Schema.Types.ObjectId, ref: 'Option' }],
});

module.exports = mongoose.model('Question', QuestionSchema);
```

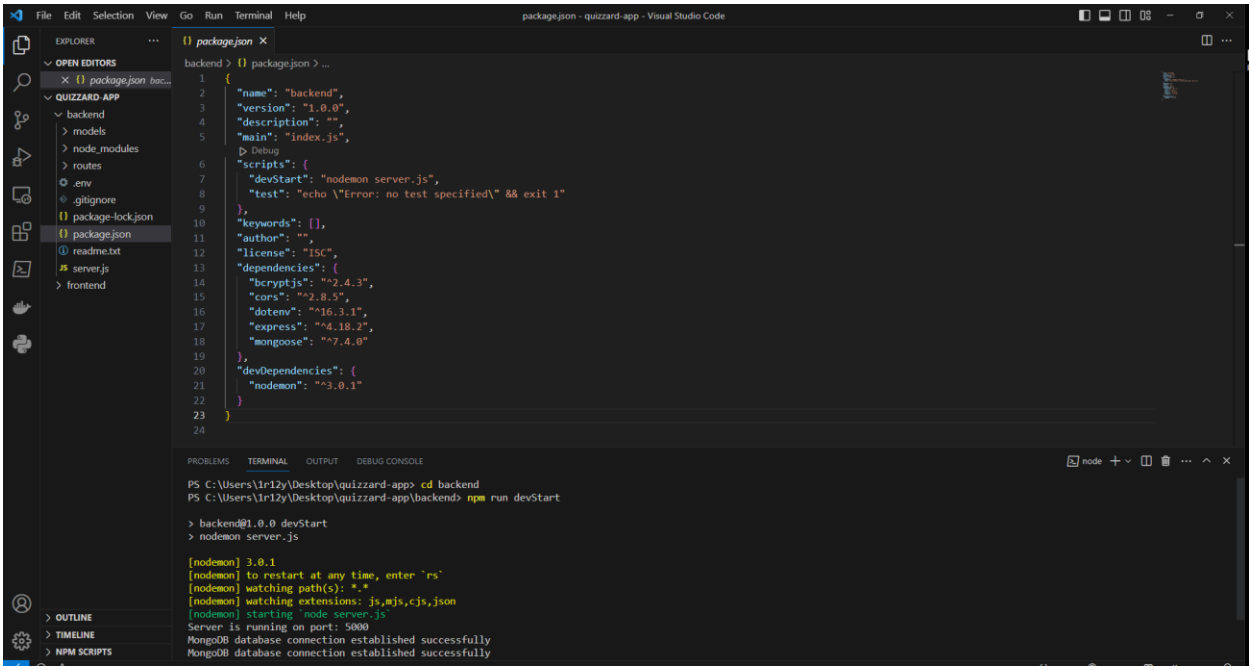
3. **Strategy Pattern:** Used for modeling the relationship between Question and Option entities.

```
option.js
backend > models > option.js > ...
//Strategy Pattern
const mongoose = require('mongoose');
const Schema = mongoose.Schema;

const OptionSchema = new Schema({
    question: { type: Schema.Types.ObjectId, ref: 'Question'
    optionText: { type: String, required: true },
    isCorrect: { type: Boolean, default: false },
});

module.exports = mongoose.model('Option', OptionSchema);
```

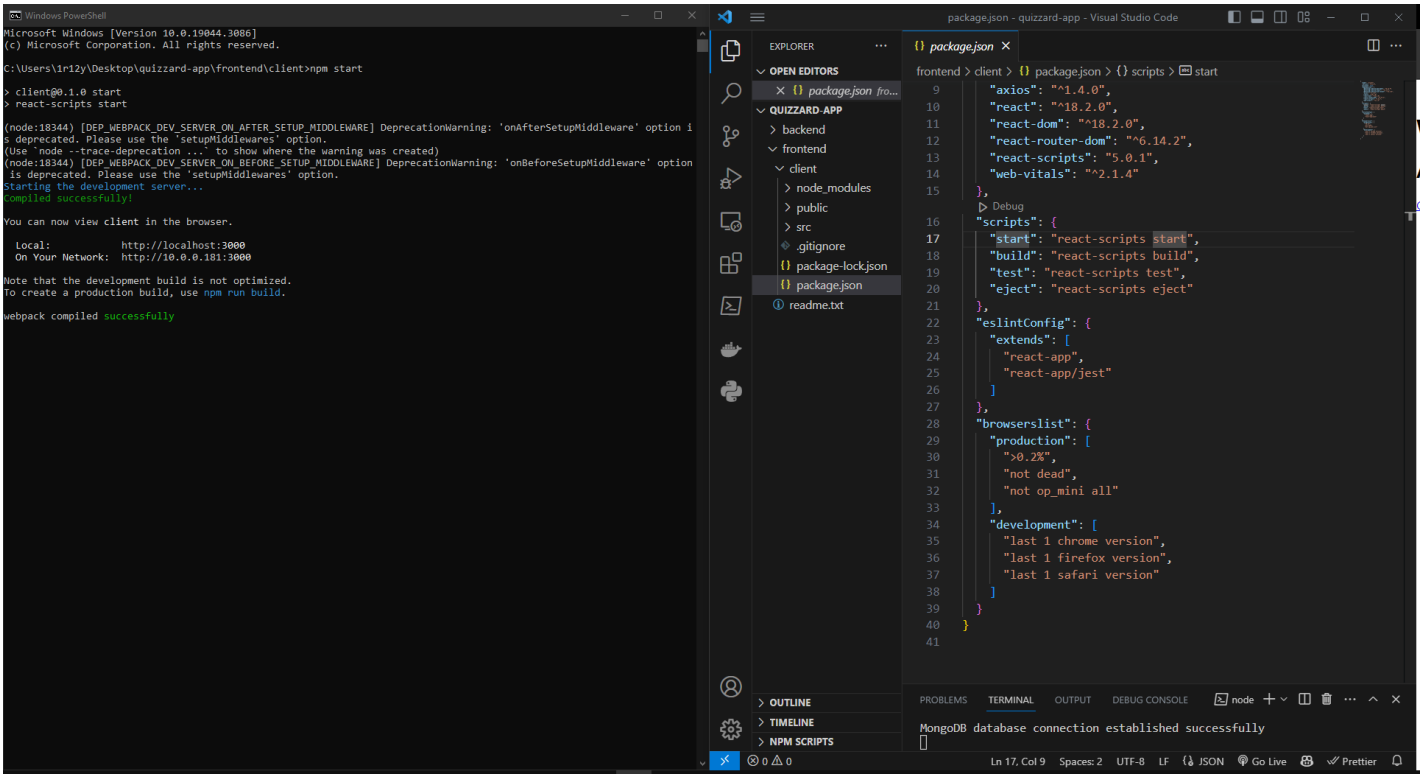
System Architecture:

At the highest level this app uses a 3-tier distributed architecture; the client(frontend), server(backend), and database. All three tiers are hosted separately but are connected via business logic.

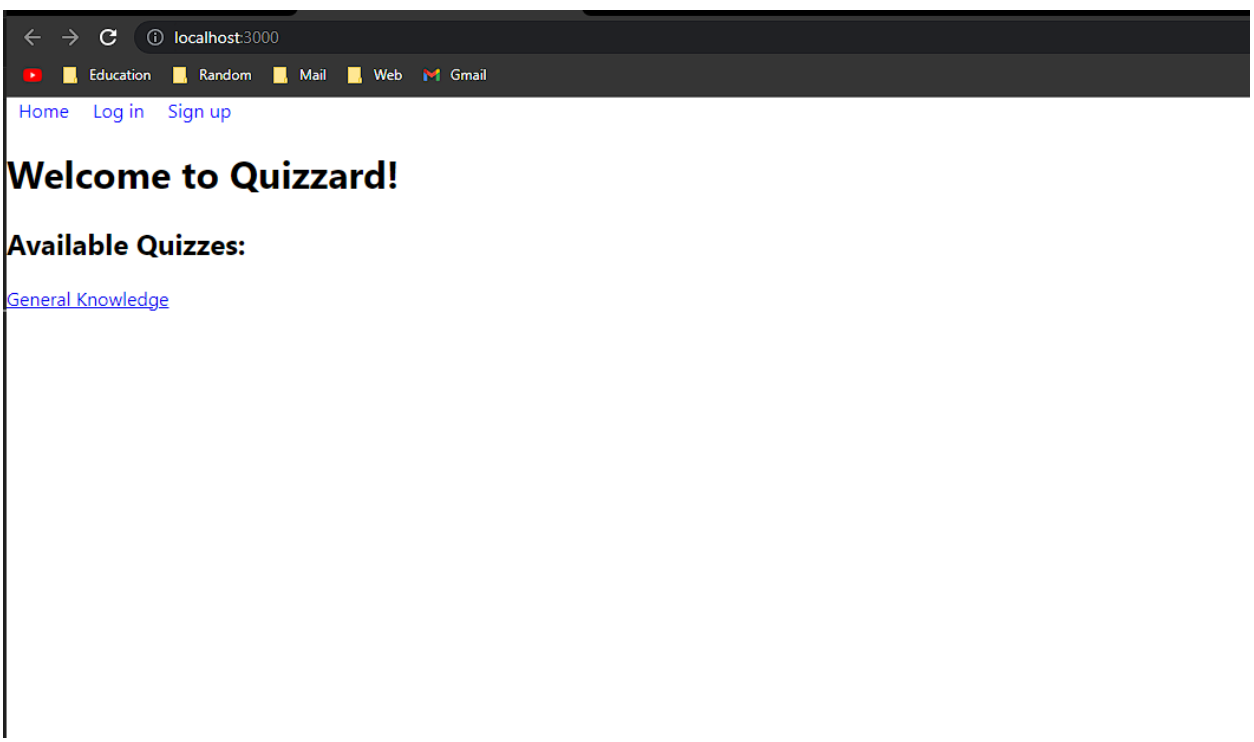
Here I’m starting up the backend on localhost port 5000 with NPM:



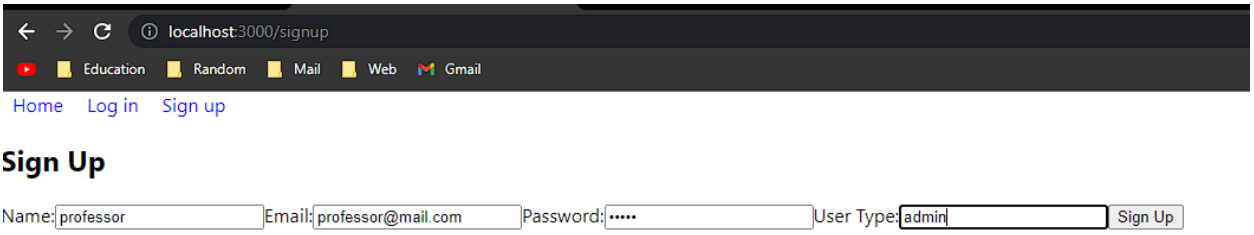
Here I’m starting up the frontend in a separate terminal on localhost port 3000 with NPM:



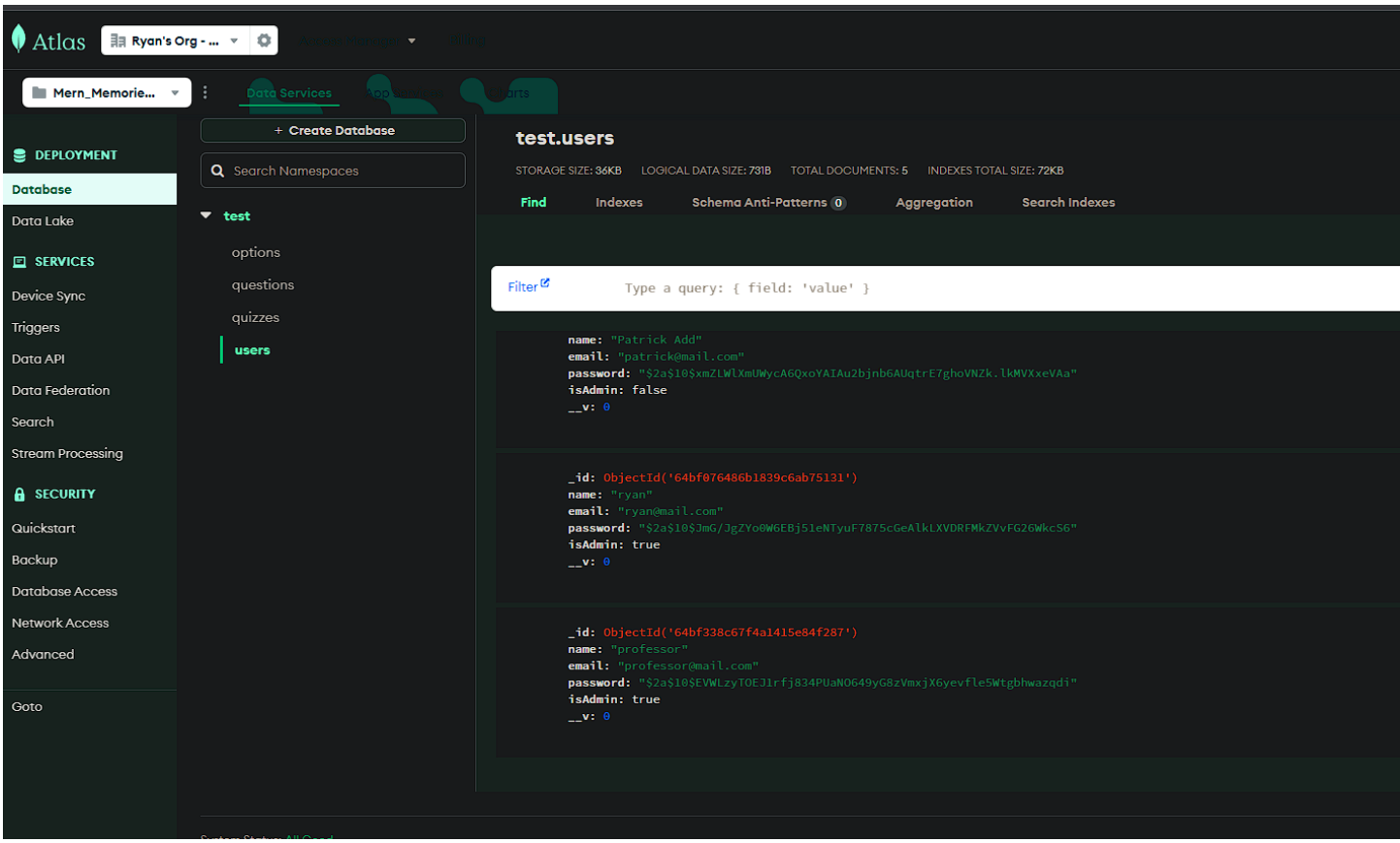
Here is the resulting React app:



Here I am creating a new Admin User:



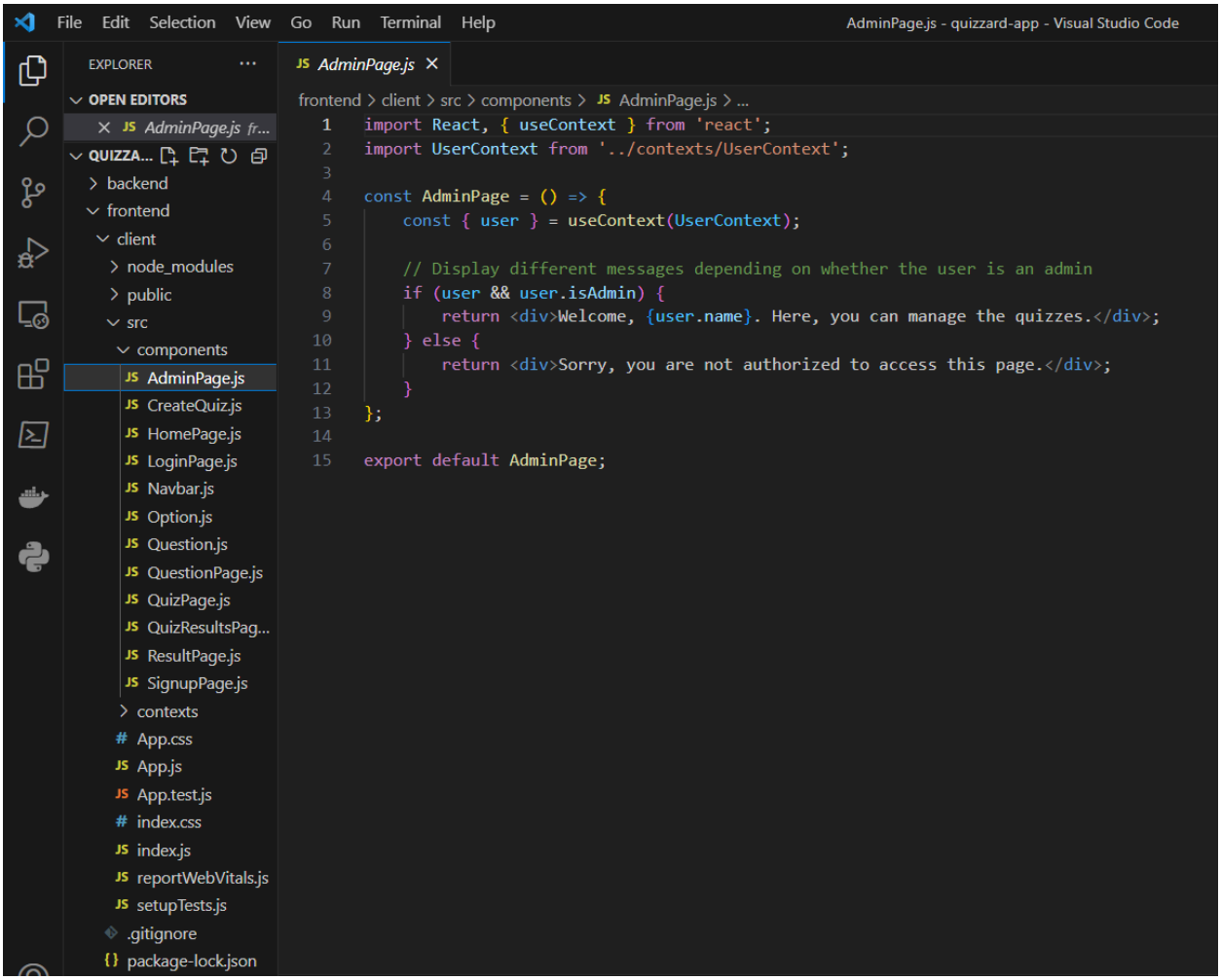
Here is the new user document in the database:



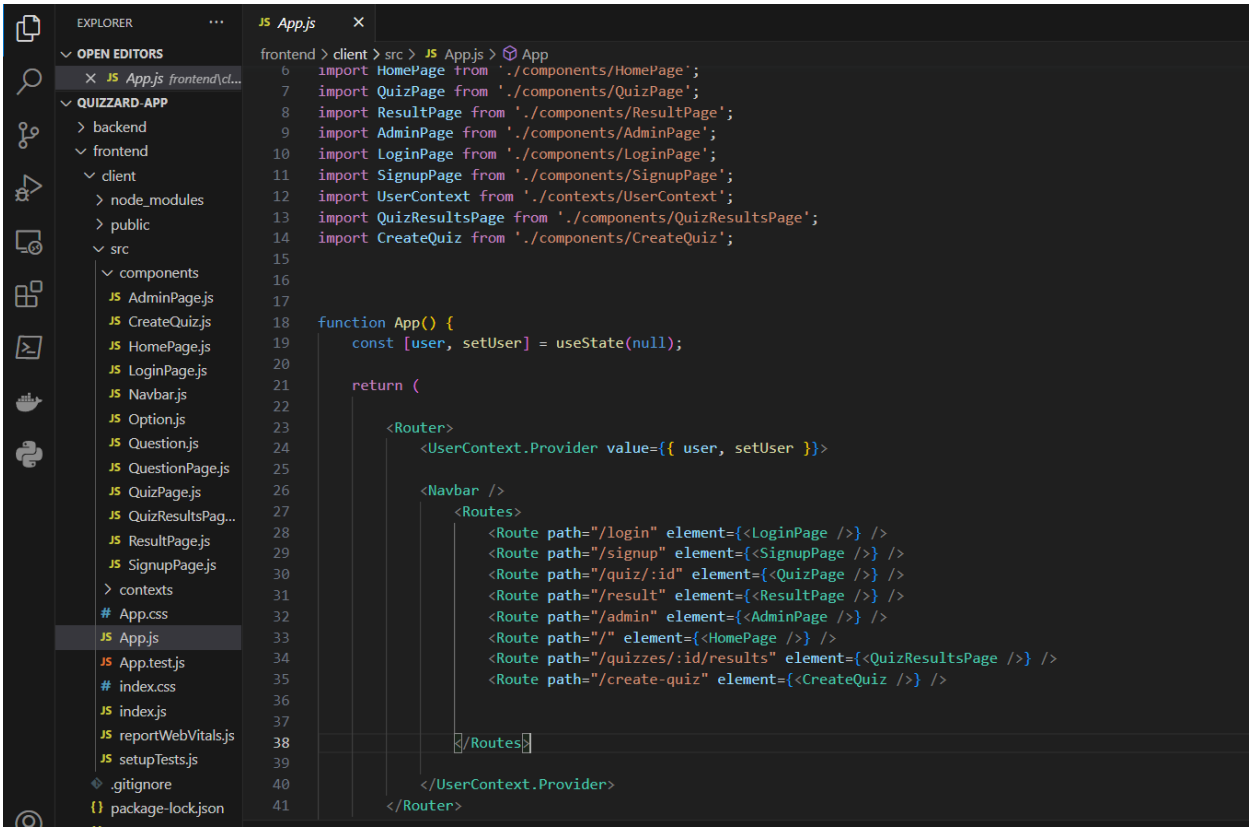
UI Architecture:

The React framework is created to use a component-based architecture. In this architecture the application is divided into reusable components. Each component is responsible for its own logic and rendering, which makes it easier to manage and maintain the codebase. These components are powered by JSX, which is an extended version of JavaScript used for rendering components and more.

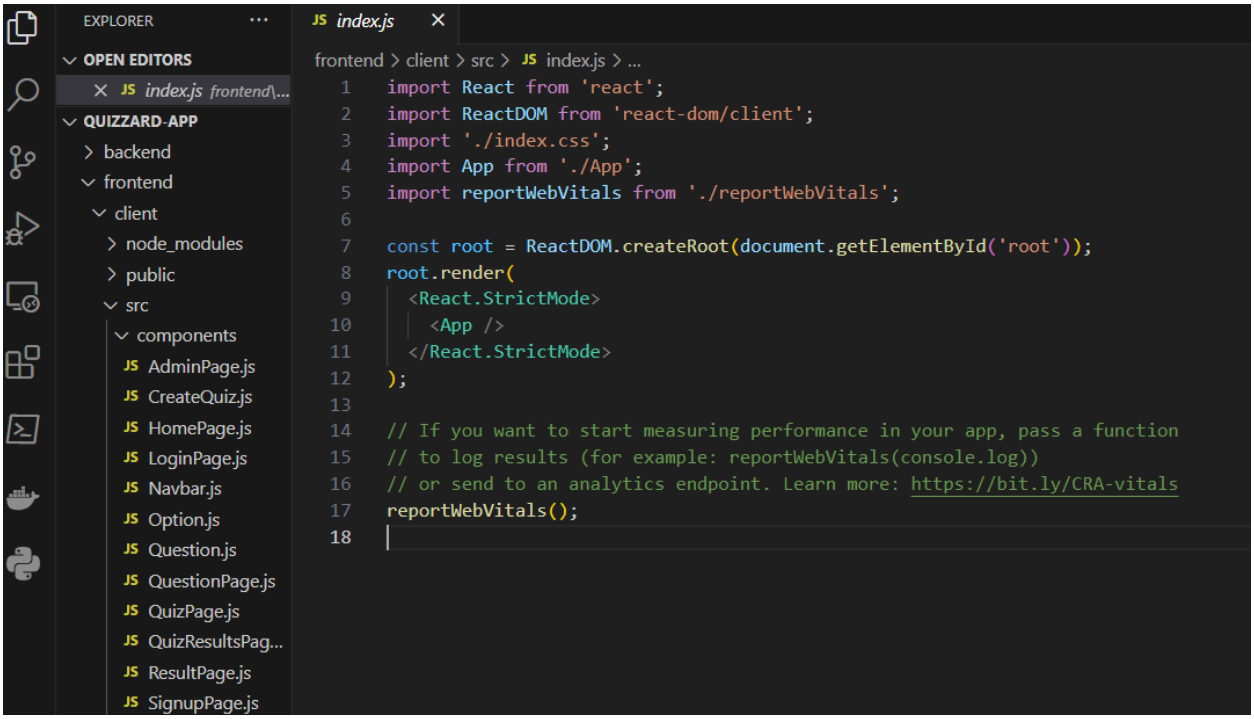
Here are the UI components:



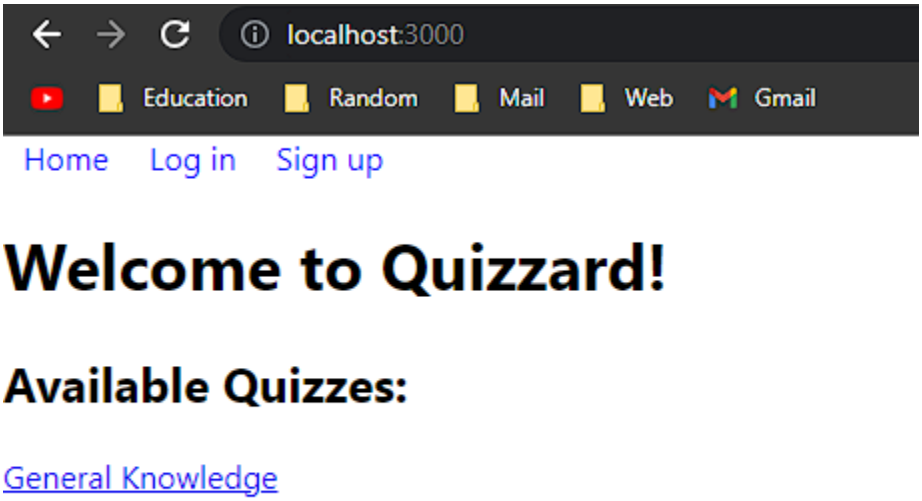
Here, logic links the components via routes which are used in the Navbar component:



Here is the App.js with all of its components being rendered:



This is the resulting home page:



Summary:

Due to time constraints I was only able to partially implement the application. Here are some of the primary features I managed to implement:

1. A frontend built with React, utilizing a component-based architecture.

2 Frontend features for users to create quizzes with questions and options, (partially implemented in the backend)

3. The frontend communicates with the backend API to store and retrieve data from the database. (It works for users currently)

4. Mostly implemented API endpoints for handling quiz and user actions.

5. Partial successful implementation of the 3 tier system; some successful communication between client, server, and database; particularly related to Users.

Pros and Cons of My Patterns:

1. Composite Pattern:
    - Pros:
        - Provides a unified interface for working with individual objects and compositions of objects.
        - Simplifies the client code by treating individual objects and compositions uniformly.
        - Allows you to work with complex nested structures of objects in a recursive manner.
        - Supports the addition and removal of objects at runtime, making it flexible for dynamic structures.
    - Cons:
        - Can lead to increased complexity, especially when the structure becomes deeply nested.
        - May not be suitable for systems with simple and straightforward object structures.
2. Strategy Pattern:
    - Pros:
        - Encapsulates algorithms or behaviors in separate classes, promoting cleaner and more maintainable code.
        - Allows for easy replacement of algorithms without modifying the context that uses them.
        - Enables better code reuse and separation of concerns by decoupling the context from specific algorithms.
        - Promotes flexibility, as new algorithms can be added without affecting existing code.
    - Cons:
        - Can introduce overhead due to the additional classes and abstractions required.

        - May lead to more complex code in cases where the number of strategies is limited or the algorithms are straightforward.

3. Factory Pattern:
    - Pros:
        - Abstracts the object creation process, making it easier to change or extend the creation logic.
        - Provides a centralized place for creating objects, which can be helpful for managing object creation complexities.
        - Encapsulates the creation logic, reducing duplication and adhering to the Single Responsibility Principle.
        - Improves code readability and maintainability by eliminating the need for complex object creation code in multiple places.
    - Cons:
        - Can lead to increased complexity for simple cases where the object creation logic is straightforward.
        - May introduce additional overhead in cases where object creation is not a critical concern.