

Ryan O'Connor

SWE 6853

Term Project

Summer 2023

### Table of contents:

Description.....	Pg. 1
Requirements.....	Pg. 1-2
Entities.....	Pg. 2
Design Patterns.....	Pgs. 3-4
System Architecture.....	Pgs. 5-7
UI Architecture.....	Pgs. 8-10
Summary.....	Pg. 11
Pros and Cons.....	Pg. 11-12

### 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

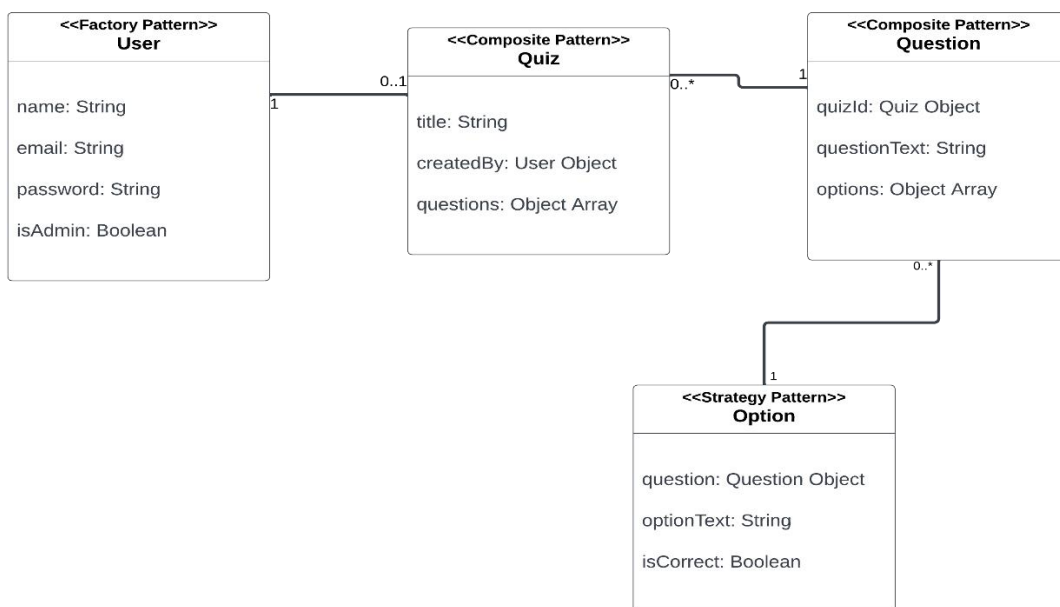
I developed this application around 4 primary entities being: User, Quiz, Question, and Option.

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 > ...
1 //Factory Pattern
2 const mongoose = require('mongoose');
3 const Schema = mongoose.Schema;
4
5 const UserSchema = new Schema({
6   name: { type: String, required: true },
7   email: { type: String, required: true, unique: true },
8   password: { type: String, required: true },
9   isAdmin: { type: Boolean, default: false },
10 });
11
12 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 > ...
1 //Composite Pattern
2 const mongoose = require('mongoose');
3 const Schema = mongoose.Schema;
4
5 const QuizSchema = new Schema({
6   title: { type: String, required: true },
7   questions: [{ type: Schema.Types.ObjectId, ref: 'Question' }],
8   createdBy: { type: Schema.Types.ObjectId, ref: 'User' },
9 });
10
11 module.exports = mongoose.model('Quiz', QuizSchema);
```

**JS** question.js ✕backend > models > **JS** question.js > ...

```
1 //Composite Pattern
2 const mongoose = require('mongoose');
3 const Schema = mongoose.Schema;
4
5 const QuestionSchema = new Schema({
6   quiz: { type: Schema.Types.ObjectId, ref: 'Quiz' },
7   questionText: { type: String, required: true },
8   options: [{ type: Schema.Types.ObjectId, ref: 'Option' }],
9 });
10
11 module.exports = mongoose.model('Question', QuestionSchema);
```

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

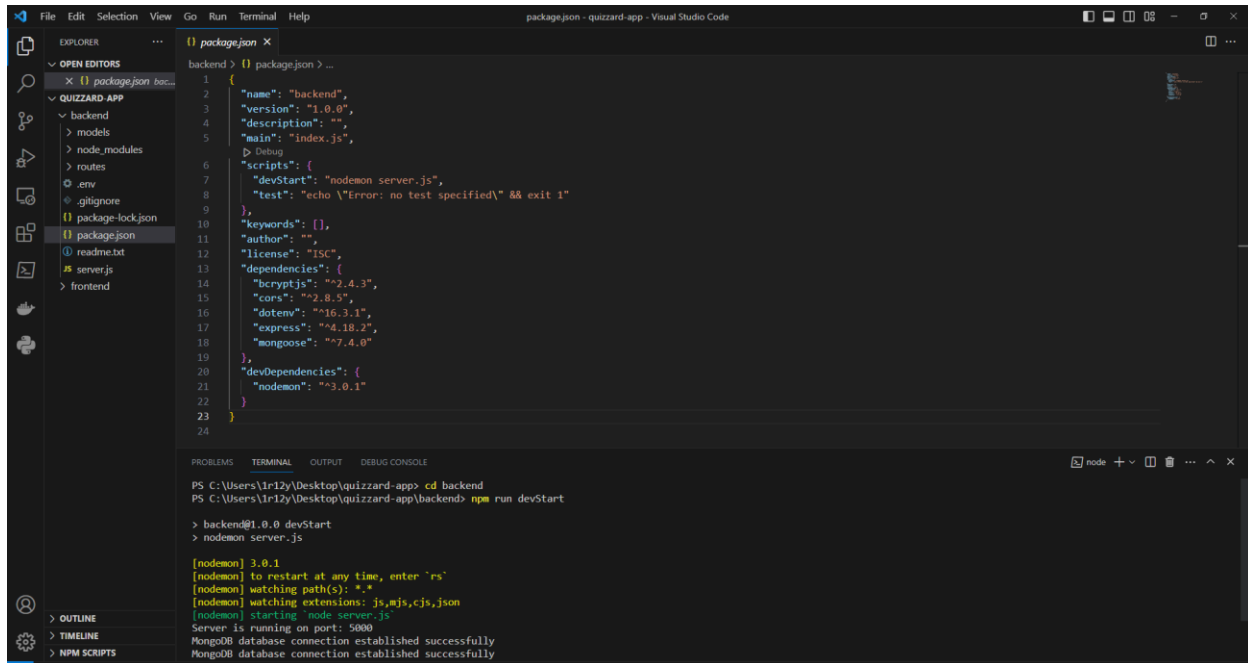
**JS** option.js ✕backend > models > **JS** option.js > ...

```
1 //Strategy Pattern
2 const mongoose = require('mongoose');
3 const Schema = mongoose.Schema;
4
5 const OptionSchema = new Schema({
6   question: { type: Schema.Types.ObjectId, ref: 'Question' },
7   optionText: { type: String, required: true },
8   isCorrect: { type: Boolean, default: false },
9 });
10
11 module.exports = mongoose.model('Option', OptionSchema);
12
```

## 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:



```
package.json
1 {
2   "name": "backend",
3   "version": "1.0.0",
4   "description": "",
5   "main": "index.js",
6   "scripts": {
7     "devStart": "nodemon server.js",
8     "test": "echo \"Error: no test specified\" && exit 1"
9   },
10  "keywords": [],
11  "author": "",
12  "license": "ISC",
13  "dependencies": {
14    "bcryptjs": "^2.4.3",
15    "cors": "^2.8.5",
16    "dotenv": "^16.3.1",
17    "express": "^4.18.2",
18    "mongoose": "^7.4.0"
19  },
20  "devDependencies": {
21    "nodemon": "^3.0.1"
22  }
23 }
24
```

```
PS C:\Users\1r12y\Desktop\quizzard-app> cd backend
PS C:\Users\1r12y\Desktop\quizzard-app\backend> npm run devStart
> backend@1.0.0 devStart
> nodemon server.js

[nodemon] 3.0.1
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting 'node server.js'
Server is running on port: 5000
MongoDB database connection established successfully
MongoDB database connection established successfully
```

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

The screenshot shows a Visual Studio Code editor with a terminal window on the left and a package.json file on the right. The terminal window displays the output of running 'npm start' in the 'client' directory. It shows deprecation warnings for 'onAfterSetupMiddleware' and 'onBeforeSetupMiddleware' options, followed by a successful compilation message: 'webpack compiled successfully'. The package.json file on the right is for the 'quizzard-app' and includes dependencies for 'axios', 'react', 'react-dom', 'react-router-dom', 'react-scripts', and 'web-vitals'. It also defines scripts for 'start', 'build', 'test', and 'eject', and includes an 'eslintConfig' section.

```

C:\Users\1r12y\Desktop\quizzard-app\frontend\client>npm start
> client@0.1.0 start
> react-scripts start

(node:18344) [DEP_WEBPACK_DEV_SERVER_ON_AFTER_SETUP_MIDDLEWARE] DeprecationWarning: 'onAfterSetupMiddleware' option is deprecated. Please use the 'setupMiddlewares' option.
(Use 'node --trace-deprecation...' to show where the warning was created)
(node:18344) [DEP_WEBPACK_DEV_SERVER_ON_BEFORE_SETUP_MIDDLEWARE] DeprecationWarning: 'onBeforeSetupMiddleware' option is deprecated. Please use the 'setupMiddlewares' option.
Starting the development server...
Compiled successfully!

You can now view client in the browser.

Local:      http://localhost:3000
On Your Network: http://10.0.0.181:3000

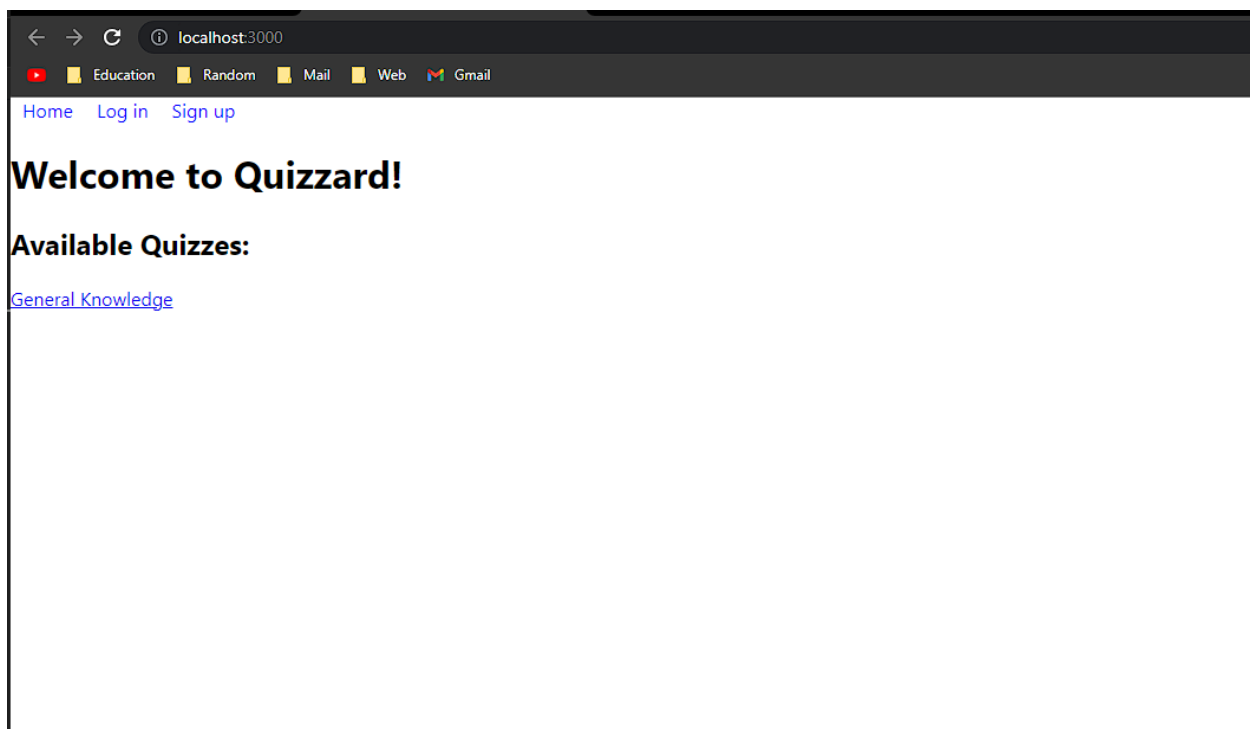
Note that the development build is not optimized.
To create a production build, use npm run build.

webpack compiled successfully
  
```

```

{
  "name": "quizzard-app",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "axios": "^1.4.0",
    "react": "^18.2.0",
    "react-dom": "^18.2.0",
    "react-router-dom": "^6.14.2",
    "react-scripts": "5.0.1",
    "web-vitals": "^2.1.4"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "eslintConfig": {
    "extends": [
      "react-app",
      "react-app/jest"
    ]
  },
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  }
}
  
```

Here is the resulting React app:



Here I am creating a new Admin User:

localhost:3000/signup

Home Log in Sign up

## Sign Up

Name:  Email:  Password:  User Type:

Here is the new user document in the database:

Atlas Ryan's Org - ...

Mern\_Memorie...

+ Create Database

Search Namespaces

test

options

questions

quizzes

users

test.users

STORAGE SIZE: 36KB LOGICAL DATA SIZE: 731B TOTAL DOCUMENTS: 5 INDEXES TOTAL SIZE: 72KB

Find Indexes Schema Anti-Patterns 0 Aggregation Search Indexes

Filter Type a query: { field: 'value' }

```

name: "Patrick Add"
email: "patrick@mail.com"
password: "$2a$10$xmZLWlXmUWycA6QxoYATu2bjnb6AUqtrE7ghoVNZk,1kMVXxeVAA"
isAdmin: false
__v: 0

_id: ObjectId('64bf076486b1839c6ab75131')
name: "ryan"
email: "ryan@mail.com"
password: "$2a$10$3mG/3gZY0W6EBj51eNTyuF7875cGeA1kLVDRFMkZVvFG26Wkc56"
isAdmin: true
__v: 0

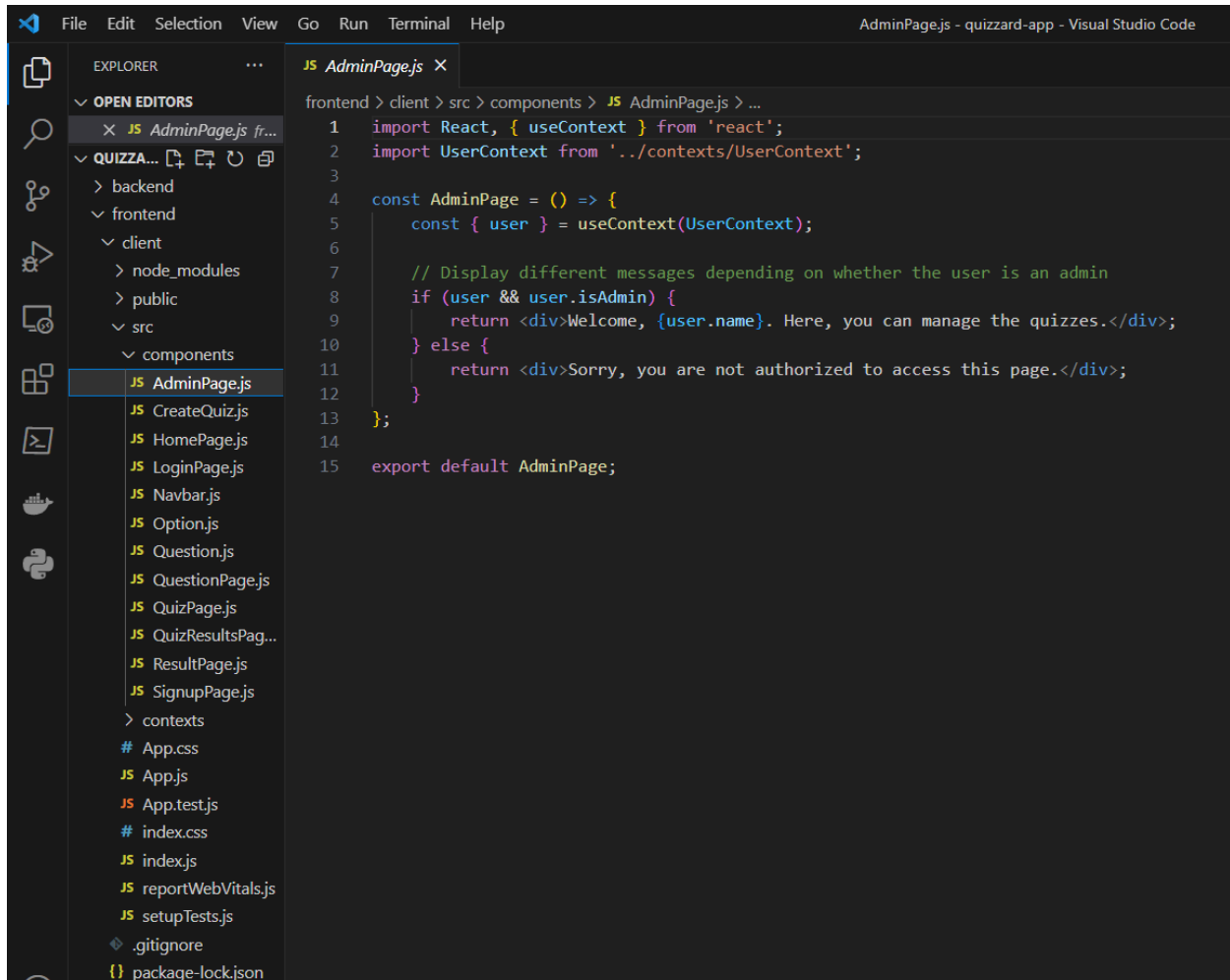
_id: ObjectId('64bf338c67f4a1415e84f287')
name: "professor"
email: "professor@mail.com"
password: "$2a$10$EVLzyT0E31rfj834PUaNO649yG8zVmxjX6yevfle5Wtghwazqdi"
isAdmin: true
__v: 0

```

## 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:



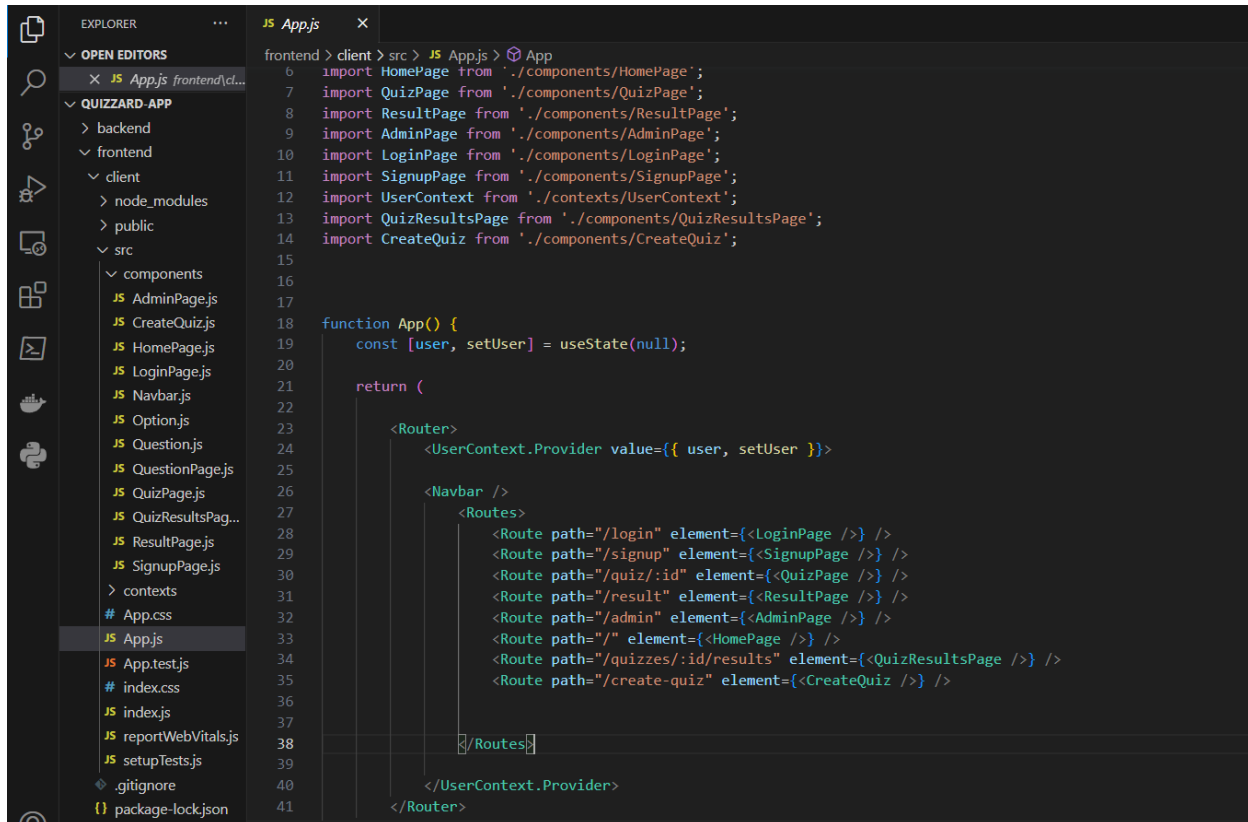
```
File Edit Selection View Go Run Terminal Help AdminPage.js - quizzard-app - Visual Studio Code

EXPLORER
OPEN EDITORS
  JS AdminPage.js fr...
QUIZZA...
  > backend
  > frontend
  > client
    > node_modules
    > public
    > src
      > components
        JS AdminPage.js
        JS CreateQuiz.js
        JS HomePage.js
        JS LoginPage.js
        JS Navbar.js
        JS Option.js
        JS Question.js
        JS QuestionPage.js
        JS QuizPage.js
        JS QuizResultsPag...
        JS ResultPage.js
        JS SignupPage.js
      > contexts
    # App.css
    JS App.js
    JS App.test.js
    # index.css
    JS index.js
    JS reportWebVitals.js
    JS setupTests.js
    .gitignore
    package-lock.json

JS AdminPage.js
frontend > client > src > components > JS AdminPage.js > ...
1 import React, { useContext } from 'react';
2 import UserContext from '../contexts/UserContext';
3
4 const AdminPage = () => {
5   const { user } = useContext(UserContext);
6
7   // Display different messages depending on whether the user is an admin
8   if (user && user.isAdmin) {
9     return <div>Welcome, {user.name}. Here, you can manage the quizzes.</div>;
10  } else {
11    return <div>Sorry, you are not authorized to access this page.</div>;
12  }
13 };
14
15 export default AdminPage;
```

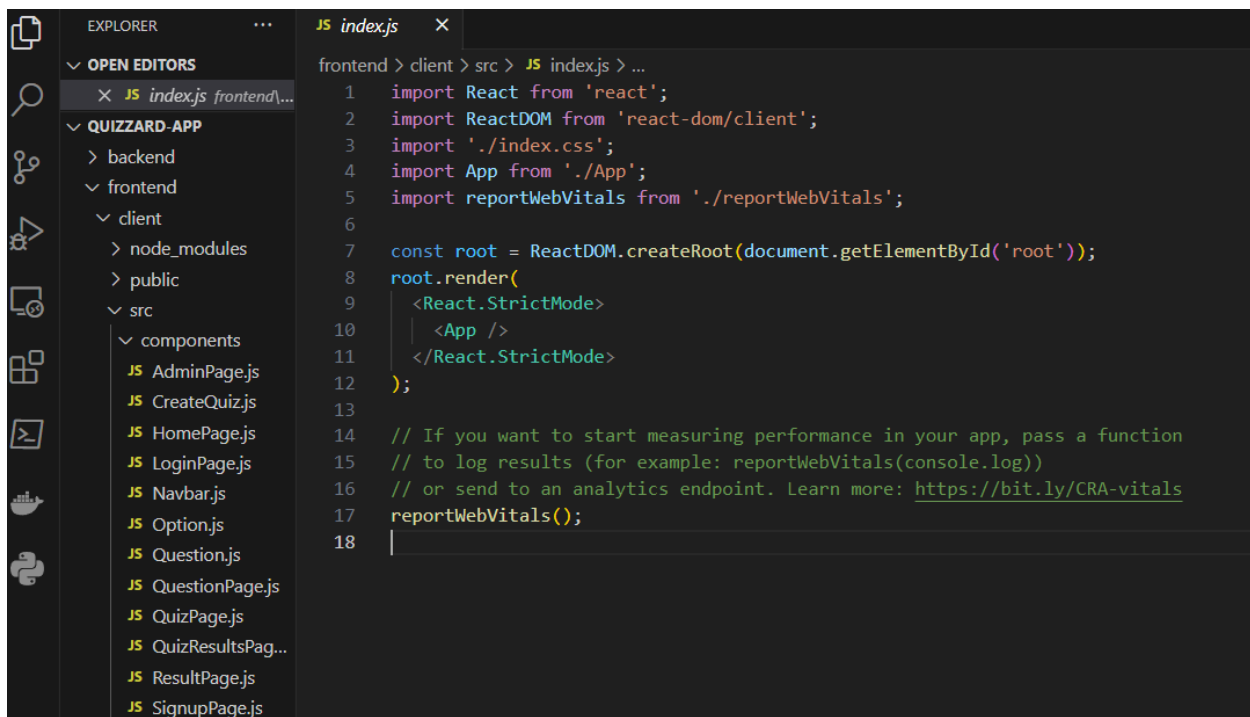


Here, logic links the components via routes which are used in the Navbar component:



```
frontent > client > src > JS App.js > App
6  import HomePage from './components/HomePage';
7  import QuizPage from './components/QuizPage';
8  import ResultPage from './components/ResultPage';
9  import AdminPage from './components/AdminPage';
10 import LoginPage from './components/LoginPage';
11 import SignupPage from './components/SignupPage';
12 import UserContext from './contexts/UserContext';
13 import QuizResultsPage from './components/QuizResultsPage';
14 import CreateQuiz from './components/CreateQuiz';
15
16
17
18 function App() {
19   const [user, setUser] = useState(null);
20
21   return (
22     <Router>
23       <UserContext.Provider value={{ user, setUser }}>
24         <Navbar />
25         <Routes>
26           <Route path="/login" element={<LoginPage />} />
27           <Route path="/signup" element={<SignupPage />} />
28           <Route path="/quiz/:id" element={<QuizPage />} />
29           <Route path="/result" element={<ResultPage />} />
30           <Route path="/admin" element={<AdminPage />} />
31           <Route path="/" element={<HomePage />} />
32           <Route path="/quizzes/:id/results" element={<QuizResultsPage />} />
33           <Route path="/create-quiz" element={<CreateQuiz />} />
34         </Routes>
35       </UserContext.Provider>
36     </Router>
37   );
38 }
39
40
41
```

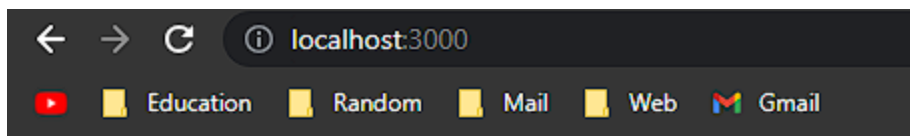
Here is the App.js with all of its components being rendered:



The screenshot shows the Visual Studio Code interface. On the left, the Explorer sidebar displays the project structure for 'QUIZZARD-APP', including folders for 'backend', 'frontend', and 'client', and a 'src' folder containing various component files like 'AdminPage.js', 'CreateQuiz.js', 'HomePage.js', 'LoginPage.js', 'Navbar.js', 'Option.js', 'Question.js', 'QuestionPage.js', 'QuizPage.js', 'QuizResultsPag...', 'ResultPage.js', and 'SignupPage.js'. The main editor area shows the 'index.js' file with the following code:

```
1 import React from 'react';
2 import ReactDOM from 'react-dom/client';
3 import './index.css';
4 import App from './App';
5 import reportWebVitals from './reportWebVitals';
6
7 const root = ReactDOM.createRoot(document.getElementById('root'));
8 root.render(
9   <React.StrictMode>
10     <App />
11   </React.StrictMode>
12 );
13
14 // If you want to start measuring performance in your app, pass a function
15 // to log results (for example: reportWebVitals(console.log))
16 // or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
17 reportWebVitals();
18
```

This is the resulting home page:



# Welcome to Quizzard!

## Available Quizzes:

[General Knowledge](#)

### 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.