Experiment 5

### **Aim**

To implement and manage **complex application state** using **Redux** and the **React Context API** in a full-stack web application.  
 The objective is to efficiently handle shared data (such as authentication, user roles, and UI states) across multiple components without prop drilling, ensuring a scalable and maintainable architecture.

### **Theory**

**1. Introduction to State Management** In React applications, state represents dynamic data that determines how components behave and render. As applications grow, managing and synchronizing state across multiple components becomes challenging.

**State management** tools like **Redux** and **Context API** help maintain a **single source of truth**, making data flow predictable and easier to debug.

**2. Redux Overview** **Redux** is a predictable state container for JavaScript applications. It manages global state using a unidirectional data flow and three core principles:

* **Single Source of Truth:** The entire app’s state is stored in one central object called the *store*.
* **State is Read-Only:** The only way to change the state is by dispatching an *action*.
* **Changes via Pure Functions:** Reducers specify how the state changes in response to actions.

**Redux Components:**

* **Store:** Holds the global state tree.
* **Action:** A plain object describing what happened.
* **Reducer:** A function determining how state changes based on an action.
* **Dispatch:** Triggers an action to update the state.

**3. Context API Overview** The **React Context API** provides a simpler way to manage global state without installing Redux.  
 It allows data to be shared between components via:

* **Context Provider:** Supplies the state and functions.
* **Context Consumer (or useContext):** Accesses the provided data.

Context is best for medium-sized apps or when global data (like theme, user authentication, or language) needs to be accessible by many components.

**4. Redux vs Context API**

| **Feature** | **Redux** | **Context API** |
| --- | --- | --- |
| Complexity | Higher (more setup) | Lower (built-in) |
| Scalability | Best for large apps | Suitable for small–medium apps |
| Performance | Optimized with middleware | Might re-render frequently |
| DevTools | Advanced debugging tools | Minimal debugging |
| Ideal For | Complex logic, async data, multi-user apps | Simple global state sharing |

### **Procedure**

#### **Step 1 – Install Redux Toolkit (Recommended)**

npm install @reduxjs/toolkit react-redux

#### **Step 2 – Create Redux Store**

Create a file: src/redux/store.js

import { configureStore } from '@reduxjs/toolkit';

import userReducer from './userSlice';

export const store = configureStore({

reducer: {

user: userReducer,

},

});

#### **Step 3 – Define a Slice (State + Reducers)**

Create src/redux/userSlice.js

import { createSlice } from '@reduxjs/toolkit';

const initialState = {

name: "Guest",

role: "candidate",

loggedIn: false,

};

const userSlice = createSlice({

name: "user",

initialState,

reducers: {

login: (state, action) => {

state.name = action.payload.name;

state.role = action.payload.role;

state.loggedIn = true;

},

logout: (state) => {

state.name = "Guest";

state.role = "candidate";

state.loggedIn = false;

},

},

});

export const { login, logout } = userSlice.actions;

export default userSlice.reducer;

#### **Step 4 – Provide the Store**

Wrap your app with the Redux <Provider> in src/index.js:

import React from 'react';

import ReactDOM from 'react-dom';

import App from './App';

import { Provider } from 'react-redux';

import { store } from './redux/store';

ReactDOM.render(

<Provider store={store}>

<App />

</Provider>,

document.getElementById('root')

);

#### **Step 5 – Use State and Dispatch in Components**

import { useSelector, useDispatch } from 'react-redux';

import { login, logout } from './redux/userSlice';

function Dashboard() {

const user = useSelector((state) => state.user);

const dispatch = useDispatch();

return (

<div className="p-4">

<h1>Welcome, {user.name}</h1>

{user.loggedIn ? (

<button onClick={() => dispatch(logout())}>Logout</button>

) : (

<button onClick={() => dispatch(login({ name: "Sanket", role: "admin" }))}>Login</button>

)}

</div>

);

}

#### **Step 6 – Implement Using Context API (Alternative)**

import React, { createContext, useContext, useState } from 'react';

const UserContext = createContext();

export function UserProvider({ children }) {

const [user, setUser] = useState({ name: "Guest", loggedIn: false });

const login = (name) => setUser({ name, loggedIn: true });

const logout = () => setUser({ name: "Guest", loggedIn: false });

return (

<UserContext.Provider value={{ user, login, logout }}>

{children}

</UserContext.Provider>

);

}

export function useUser() {

return useContext(UserContext);

}

Usage:

function Profile() {

const { user, login, logout } = useUser();

return (

<div>

<h2>User: {user.name}</h2>

{user.loggedIn ? (

<button onClick={logout}>Logout</button>

) : (

<button onClick={() => login("Sanket")}>Login</button>

)}

</div>

);

}

Wrap <App /> in <UserProvider> to make the context global.

In the **Interview Simulator Website**, Redux can manage global states like:

* **User authentication and roles**
* **Active interview sessions**
* **Live chat and question data**

While the **Context API** can handle smaller global states such as theme toggling or UI preferences.

By implementing these state management techniques, the project achieves **predictable data flow**, **better scalability**, and **simplified debugging**, aligning with modern React architecture and DevOps-ready design principles.

Code

import json

def handler(request, context):

"""

Simple serverless function handler

"""

if request.method == "GET":

return {

"statusCode": 200,

"headers": {

"Content-Type": "application/json"

},

"body": json.dumps({"message": "API is online"})

}

elif request.method == "POST":

try:

body = json.loads(request.body)

except:

body = {}

return {

"statusCode": 200,

"headers": {

"Content-Type": "application/json"

},

"body": json.dumps({"status": "success", "message": "Received data"})

}

const notFound = (req, res, next) => {

const error = new Error(`Not Found - ${req.originalUrl}`);

res.status(404);

next(error);

};

const errorHandler = (err, req, res, next) => {

let statusCode = res.statusCode === 200 ? 500 : res.statusCode;

let message = err.message;

// Mongoose bad ObjectId

if (err.name === 'CastError' && err.kind === 'ObjectId') {

statusCode = 404;

message = 'Resource not found';

}

// Mongoose duplicate key

if (err.code === 11000) {

statusCode = 400;

message = 'Duplicate field value entered';

}

// Mongoose validation error

if (err.name === 'ValidationError') {

statusCode = 400;

message = Object.values(err.errors).map(val => val.message).join(', ');

}

// JWT errors

if (err.name === 'JsonWebTokenError') {

statusCode = 401;

message = 'Invalid token';

}

if (err.name === 'TokenExpiredError') {

statusCode = 401;

message = 'Token expired';

}

res.status(statusCode).json({

message,

stack: process.env.NODE\_ENV === 'production' ? '🥞' : err.stack,

error: process.env.NODE\_ENV === 'production' ? {} : err,

});

};

module.exports = { notFound, errorHandler };

const jwt = require('jsonwebtoken');

const asyncHandler = require('express-async-handler');

const User = require('../models/User');

// Protect routes - check for valid JWT token

const protect = asyncHandler(async (req, res, next) => {

let token;

if (req.headers.authorization && req.headers.authorization.startsWith('Bearer')) {

try {

// Get token from header

token = req.headers.authorization.split(' ')[1];

// Verify token

const decoded = jwt.verify(token, process.env.JWT\_SECRET);

// Get user from the token (excluding password)

req.user = await User.findById(decoded.id).select('-password');

if (!req.user) {

res.status(401);

throw new Error('Not authorized, user not found');

}

if (!req.user.isActive) {

res.status(401);

throw new Error('Account is deactivated');

}

next();

} catch (error) {

console.error('Token verification error:', error);

res.status(401);

throw new Error('Not authorized, token failed');

}

}

if (!token) {

res.status(401);

throw new Error('Not authorized, no token provided');

}

});

// Admin role check

const admin = asyncHandler(async (req, res, next) => {

if (req.user && req.user.role === 'admin') {

next();

} else {

res.status(403);

throw new Error('Not authorized as admin');

}

});

// Rate limiting middleware (basic implementation)

const rateLimit = (maxRequests = 100, windowMs = 15 \* 60 \* 1000) => {

const requests = new Map();

return (req, res, next) => {

const key = req.ip;

const now = Date.now();

if (!requests.has(key)) {

requests.set(key, { count: 1, resetTime: now + windowMs });

return next();

}

const request = requests.get(key);

if (now > request.resetTime) {

request.count = 1;

request.resetTime = now + windowMs;

return next();

}

if (request.count >= maxRequests) {

return res.status(429).json({

message: 'Too many requests, please try again later.',

retryAfter: Math.ceil((request.resetTime - now) / 1000)

});

}

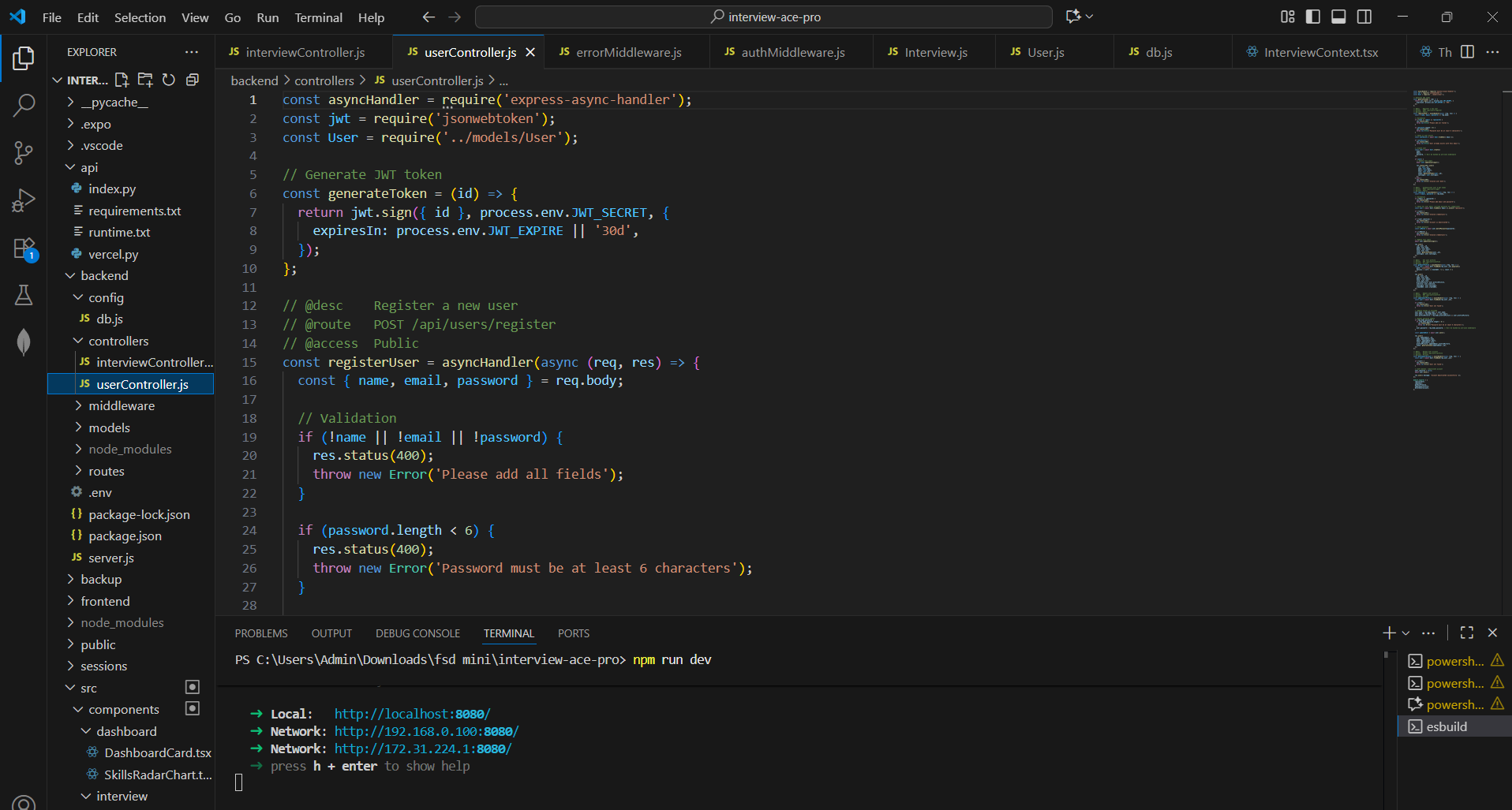
request.count++;

next();

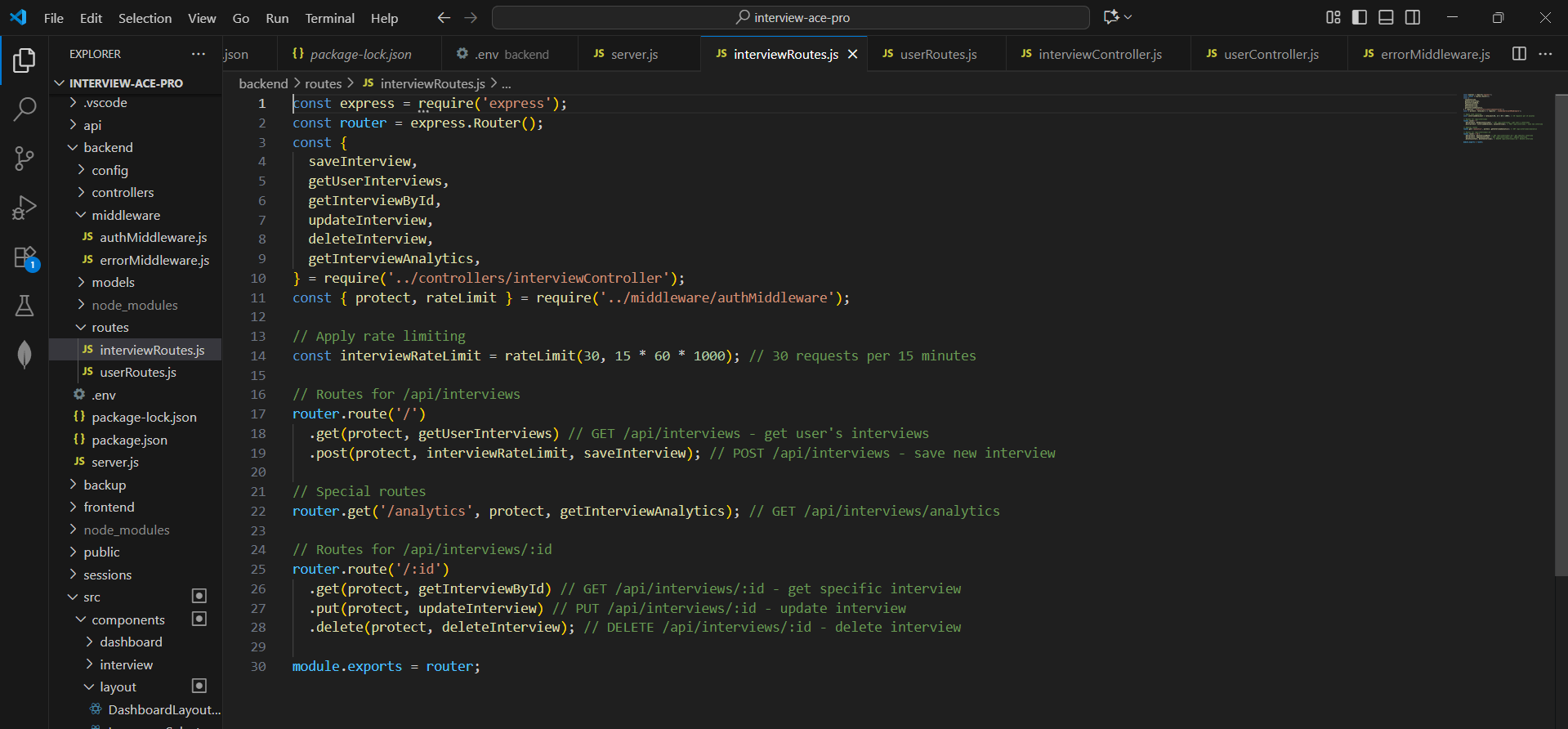
};

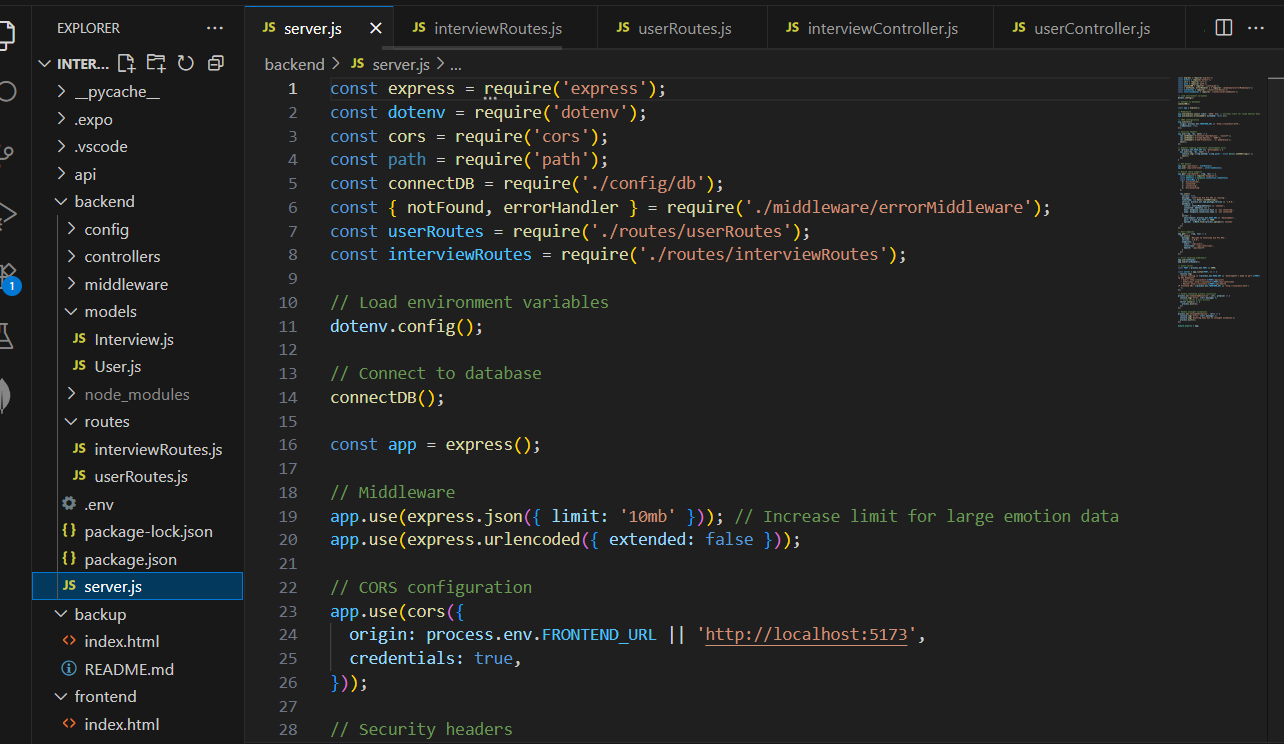
};

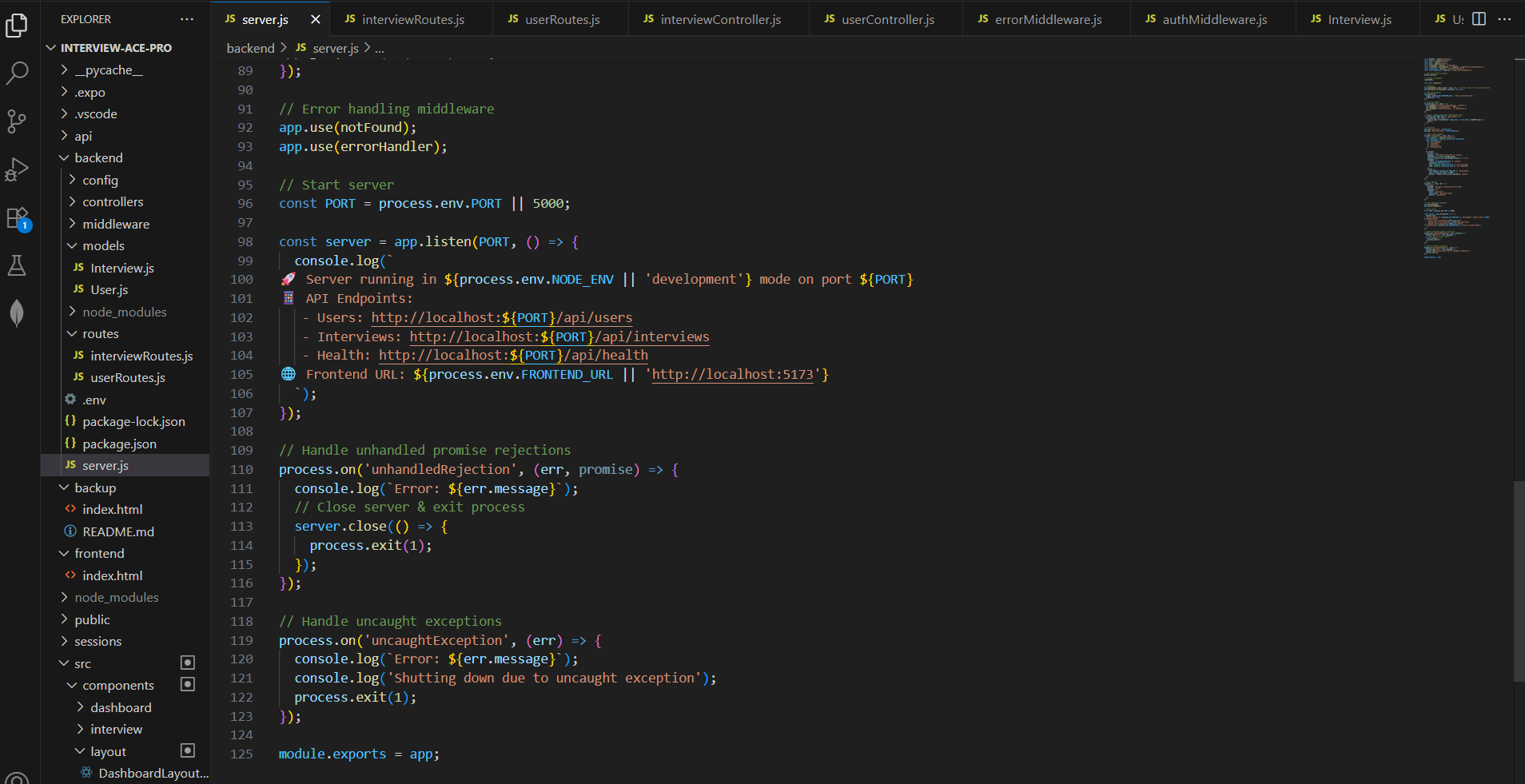
module.exports = { protect, admin, rateLimit };



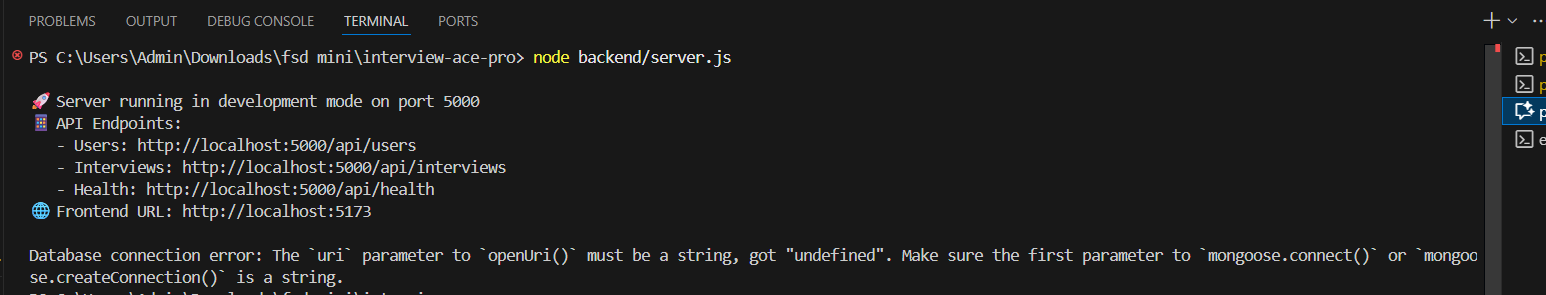
### 



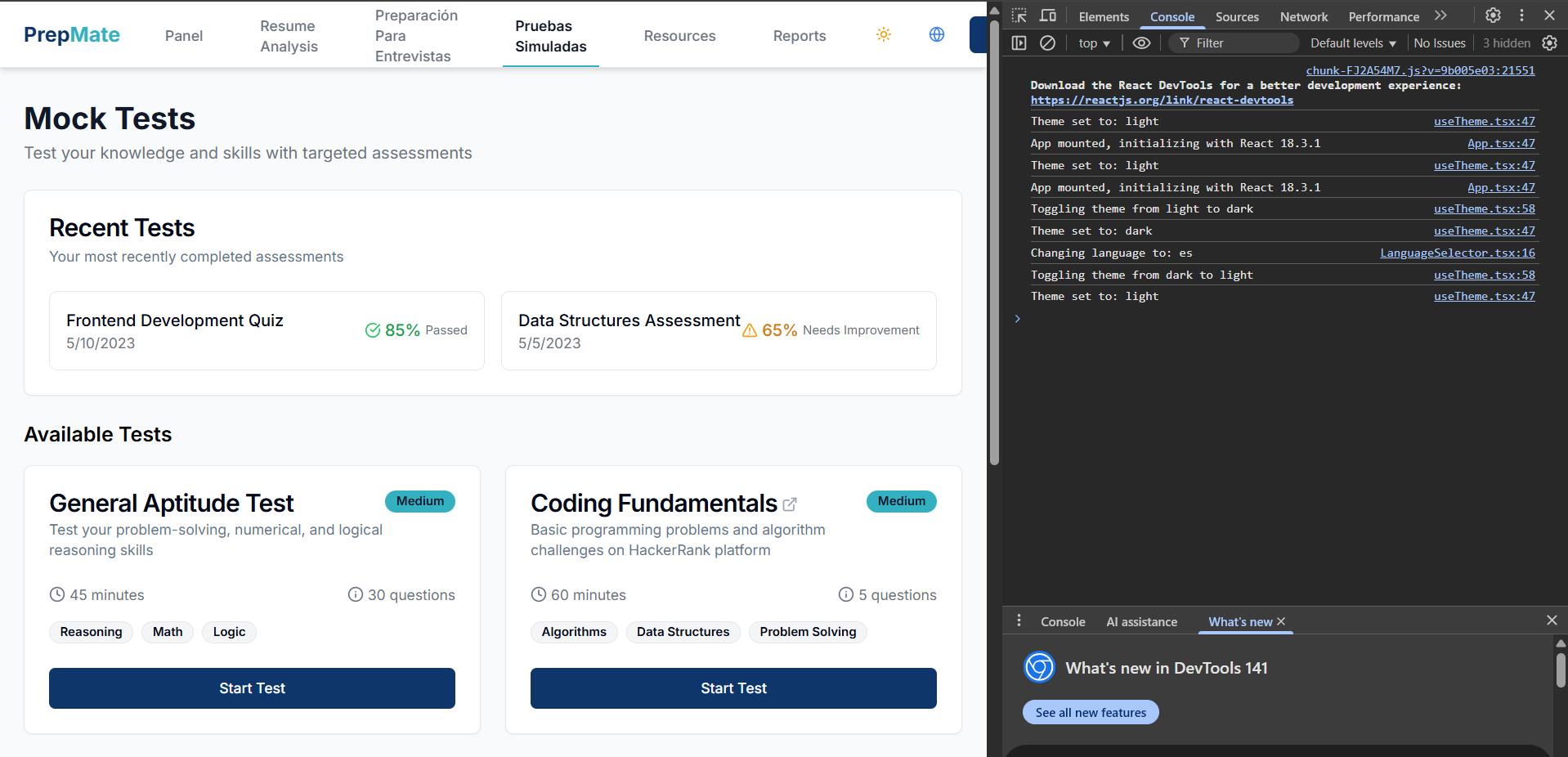
Main server .js for api calling : -



Server starting



API calling for carrying out different tasks( dark theme , course form , error handling)



### **Conclusion**

This experiment successfully demonstrated the use of **Redux** and **Context API** for managing complex global state in React.  
 Both approaches provided a structured way to share data and handle updates across multiple components without prop drilling.