**Module – 5**

**Backend**

**Index.js**

1. **Imports necessary modules:**
   * **express**: Web application framework for Node.js.
   * **dotenv**: Loads environment variables from a **.env** file.
   * **cors**: Middleware for handling Cross-Origin Resource Sharing.
2. **Connects to the database:**
   * Assumes that there is a file named **dbconnect** (imported using **require('./dbconnect')**) that contains logic for connecting to the database.
3. **Creates an Express application:**
   * Initializes an instance of the Express application using **express()** and assigns it to the variable **app**.
4. **Configures the application:**
   * Loads environment variables from a **.env** file using **dotenv.config()**.
   * Configures the application to parse JSON requests using **express.json()**.
   * Enables Cross-Origin Resource Sharing (CORS) using **app.use(cors())**.
5. **Defines routes for user and sales operations:**
   * Routes related to user operations are defined under the path **/api/v1/user**, and these routes are handled by a file assumed to be named **user\_route.js**.
   * Routes related to sales operations are defined under the path **/api/v1/sales**, and these routes are handled by a file assumed to be named **sales\_route.js**.
6. **Starts the server:**
   * Listens on the port specified in the **PORT** environment variable.
   * Logs a message to the console indicating that the server is running.

**dbconnect.js**

1. **Importing Modules:**
   * Imports the **mongoose** library for MongoDB interactions.
   * Imports **dotenv** for loading environment variables from a **.env** file.
2. **Loading Environment Variables:**
   * Calls **dotenv.config()** to load environment variables from a **.env** file.
3. **Connecting to MongoDB:**
   * Uses **mongoose.connect()** to establish a connection to the MongoDB database.
   * The database URL is retrieved from the **MONGODB\_URL** environment variable.
   * If the connection is successful, a success message is logged to the console.
   * If there's an error during the connection process, an error message is logged to the console.

**.env**

1. It contains port, database url and jwt secret.

**Routes**

**user\_route.js**

1. **Importing Modules:**
   * Imports the **express** module for creating an Express router.
   * Imports functions (**addUser** and **loginUser**) from the user controller assumed to be defined in **'../controllers/user\_controller'**.
2. **Creating an Express Router:**
   * Creates an instance of an Express router using **express.Router()**.
3. **Defining Routes:**
   * Defines two routes:
     + **POST /login**: This route is associated with the **loginUser** function from the user controller. It handles user login.
     + **POST /register**: This route is associated with the **addUser** function from the user controller. It handles user registration.
4. **Exporting the Router:**
   * Exports the router instance to make it available for use in other parts of the application.

**sales\_routes.js**

1. **Importing Modules:**
   * Imports the **express** module for creating an Express router.
   * Imports functions (**addSales**, **topSalesToday**, **totalRevenue**) from the sales controller assumed to be defined in **'../controllers/sales\_controller'**.
   * Imports a middleware (**authenticate**) assumed to be defined in **'../middlewares/protectedRoute'** for authentication.
2. **Creating an Express Router:**
   * Creates an instance of an Express router using **express.Router()**.
3. **Defining Routes:**
   * Defines three routes:
     + **POST /addSales**: This route is associated with the **addSales** function from the sales controller and is protected by the **authenticate** middleware. It handles the addition of sales.
     + **GET /topSales**: This route is associated with the **topSalesToday** function from the sales controller and is protected by the **authenticate** middleware. It retrieves today's top sales.
     + **GET /totalRevenue**: This route is associated with the **totalRevenue** function from the sales controller and is protected by the **authenticate** middleware. It retrieves the total revenue.
4. **Exporting the Router:**
   * Exports the router instance to make it available for use in other parts of the application.

**Models**

**user\_model.js**

1. **Importing Modules:**
   * Imports the **mongoose** module for working with MongoDB.
2. **Creating a Mongoose Schema:**
   * Creates a Mongoose schema for the 'User' model using **mongoose.Schema**.
   * Defines fields for the user schema: **firstName**, **lastName**, **email**, and **password**.
   * Specifies that all fields are required.
3. **Including Timestamps:**
   * Adds **{ timestamps: true }** as an option when creating the schema to include **createdAt** and **updatedAt** fields.
4. **Creating a Mongoose Model:**
   * Creates a Mongoose model named **UserModel** based on the defined user schema.
5. **Exporting the User Model:**
   * Exports the **UserModel** to make it available for use in other parts of the application.

**sales\_model.js**

1. **Importing Modules:**
   * Imports the **mongoose** module for working with MongoDB.
   * Destructures **ObjectId** from **mongoose.Schema.Types**.
2. **Creating a Mongoose Schema:**
   * Creates a Mongoose schema for the 'Sale' model using **mongoose.Schema**.
   * Defines fields for the sale schema: **productName**, **quantity**, **amount**, and **author**.
   * Specifies that **productName**, **quantity**, and **amount** are required.
   * Includes a reference (**ref: "UserModel"**) to the 'UserModel' for the 'author' field.
3. **Including Timestamps:**
   * Adds **{ timestamps: true }** as an option when creating the schema to include **createdAt** and **updatedAt** fields.
4. **Creating a Mongoose Model:**
   * Creates a Mongoose model named **SaleModel** based on the defined sale schema.
5. **Exporting the Sale Model:**
   * Exports the **SaleModel** to make it available for use in other parts of the application.

**Middlewares**

**protectedRoute.js**

1. **Importing Modules:**
   * Imports **jwt** for working with JSON Web Tokens.
   * Imports **UserModel** for interacting with user data.
   * Imports **dotenv** for loading environment variables.
   * Requires **mongoose** for MongoDB operations.
2. **Middleware Function (authenticate):**
   * The middleware function is asynchronous (**async**).
   * Extracts the authorization header from the incoming request (**req.headers["authorization"]**).
   * Checks if the authorization header is missing and responds with a 401 status if it is.
   * Extracts the token from the authorization header (**authHeader.replace('Bearer ', "")**).
   * Checks if the token is missing and responds with a 401 status if it is.
   * Attempts to verify the JWT token using **jwt.verify** and the secret from the environment variables.
   * Finds the user in the database based on the decoded token's email, excluding the password (**await UserModel.find**).
   * Responds with a 401 status if the user is not found.
   * Adds the user information to the request object (**req.user**).
   * Calls the **next()** function to proceed to the next middleware or route handler.
3. **Exporting Middleware:**
   * Exports the authentication middleware (**authenticate**) for use in other parts of the application.

**Controllers**

**user\_controller.js**

1. **Importing Modules:**
   * Imports **UserModel** for interacting with user data.
   * Imports **bcrypt** for password hashing.
   * Imports **jwt** for working with JSON Web Tokens.
   * Imports **dotenv** for loading environment variables.
2. **Controller Function (addUser):**
   * Handles user registration.
   * Destructures user input (firstName, lastName, email, password) from the request body.
   * Checks if all required fields are provided, responding with a 400 status if any are missing.
   * Checks if the email is already registered, responding with a message if it is.
   * Hashes the provided password using bcrypt.
   * Creates a new user instance with hashed password and saves it to the database.
   * Responds with a success message and user details.
3. **Controller Function (loginUser):**
   * Handles user login.
   * Destructures user input (email, password) from the request body.
   * Checks if email and password are provided, responding with a 400 status if any are missing.
   * Finds the user based on the provided email.
   * Responds with a 400 status if the user is not found.
   * Compares the provided password with the hashed password in the database.
   * Responds with a 400 status if the passwords don't match.
   * Generates a JWT token for authentication using the user's email and the JWT secret.
   * Extracts essential user information for the response.
   * Responds with a success message, token, and user information.
4. **Exporting Controller Functions:**
   * Exports the user controller functions (**addUser** and **loginUser**) for use in other parts of the application.

**sales\_controller.js**

1. **Importing Module:**
   * Imports **SaleModel** for interacting with sales data.
2. **Controller Function (addSales):**
   * Handles the addition of sales.
   * Destructures sales input (productName, quantity, amount) from the request body.
   * Checks if all required fields are provided, responding with a 400 status if any are missing.
   * Creates a new sale instance with the author (user) information.
   * Saves the new sale to the database.
   * Responds with a success message and sale details.
3. **Controller Function (topSalesToday):**
   * Handles the retrieval of the top sales for the current day.
   * Retrieves the user ID from the request object.
   * Determines the start and end of the current day in UTC.
   * Finds the top 5 sales of the user for the current day, sorted by amount in descending order.
   * Responds with a success message and the top sales of the day.
4. **Controller Function (totalRevenue):**
   * Handles the calculation of total revenue.
   * Retrieves the user ID from the request object.
   * Finds all sales of the user.
   * Calculates the total amount by summing up the amounts of all sales.
   * Responds with a success message and the total revenue.
5. **Exporting Controller Functions:**
   * Exports the sales controller functions (**addSales**, **topSalesToday**, and **totalRevenue**) for use in other parts of the application.

**Frontend**

**Index.js**

1. Install react-toastify in terminal and import it in index.js and import the Toast Container for notification.
2. Import provider for redux store.

**Config.js**

1. **Base URL:**

**REACT\_APP\_API\_BASE\_URL:** Represents the base URL for the API. In this case, it's set to <http://localhost:5000.>

1. **User-Related API Endpoint:**

**REACT\_APP\_API\_USER:** Represents the API endpoint for user-related operations. It is appended to the base URL, forming the complete URL for user-related requests (e.g., registration, login).

<http://localhost:5000/api/v1/user>

1. **Sales-Related API Endpoint:**

**REACT\_APP\_API\_SALE:** Represents the API endpoint for sales-related operations. Similar to the user endpoint, it is appended to the base URL to form the complete URL for sales-related requests (e.g., adding sales, retrieving sales data).

<http://localhost:5000/api/v1/sales>

**App.js**

1. **Import Statements:**
   * The code imports necessary styles, components, and routing-related elements from the 'react-router-dom' library.
   * Components imported include pages such as **AddSales**, **TopSales**, **Login**, **Register**, **Homepage**, and **TotalRevenue**.
2. **Routing Setup:**
   * The **App** component uses the **BrowserRouter** from 'react-router-dom' to set up client-side navigation.
   * Inside the **Router** component, there's a **Routes** component that defines different routes for various pages.
3. **Routes Configuration:**
   * Each **Route** element within the **Routes** component maps a specific URL path to a corresponding React component.
   * Example Routes:
     + **/**: Maps to the **Homepage** component.
     + **/register**: Maps to the **Register** component.
     + **/login**: Maps to the **Login** component.
     + **/addSales**: Maps to the **AddSales** component.
     + **/topSales**: Maps to the **TopSales** component.
     + **/totalRevenue**: Maps to the **TotalRevenue** component.
4. **Rendering:**
   * The **App** component returns a JSX structure that includes the **Router** with defined **Routes**.

**Redux**

**userReducer.js**

1. **Initial State:**
   * The initial state for the 'user' slice is defined as an object with an empty 'user' property.
2. **Reducer Function:**
   * The **userReducer** function is exported as a constant.
   * It takes two parameters: **state** (current state) and **action** (dispatched action).
3. **Switch Statement:**
   * The reducer uses a **switch** statement to handle different action types dispatched to the store.
4. **Action Types:**
   * **"LOGIN\_SUCCESS":**
     + Updates the state by spreading the current state (**...state**) and replacing the 'user' property with the 'payload' from the action.
     + This is typically dispatched when a user successfully logs in.
   * **"LOGIN\_ERROR":**
     + Resets the state to the initial state (**initialState**).
     + This is dispatched in case of a login error.
   * **Default Case:**
     + Returns the current state if the action type is not recognized.

**combineReducer.js**

1. **Import Statements:**
   * Imports the **combineReducers** function from the 'redux' library.
   * Imports the **userReducer** from a file named 'userReducer'.
2. **Combine Reducers:**
   * Defines a constant named **combineReducer** that holds the combined reducer.
   * Uses **combineReducers** to combine multiple reducers into a single root reducer.
   * In this case, it combines the 'userReducer', and the resulting state will have a 'userReducer' property.
3. **Root Reducer Structure:**
   * The root reducer structure is an object where keys correspond to different slices of the state managed by individual reducers.
   * In this case, it includes a 'userReducer' property.
4. **Export:**
   * Exports the combined reducer (**combineReducer**) as the default export.

**Store.js**

1. **Import Statements:**
   * Imports the **createStore** function from the 'redux' library.
   * Imports the combined reducer (**combineReducer**) from a file named 'combineReducer'.
2. **Create Redux Store:**
   * Defines a constant named **store** that holds the Redux store.
   * Uses the **createStore** function and passes the combined reducer (**combineReducer**) as an argument to create the store.
3. **Store Configuration:**
   * The **combineReducer** is passed to the **createStore** function, creating a store that holds the complete state structure defined by the combined reducer.
4. **Export:**
   * Exports the created Redux store (**store**) for use in the application.

**Pages**

**TotalRevenue.js**

1. **Component Structure:**
   * Imports necessary React components and hooks (e.g., **useEffect**, **useState**) from React and third-party libraries.
   * Imports components and functions needed for this specific component, such as **Navbar** and **toast** for notifications.
   * Utilizes Redux hooks (**useSelector**) to access user data from the Redux state.
   * Utilizes React Router's **useNavigate** hook for navigation.
2. **State and Variables:**
   * Defines a state variable **Revenue** using the **useState** hook to store the total revenue.
3. **Request Configuration:**
   * Defines a request configuration (**reqConfig**) with an Authorization header containing the user's token.
4. **Fetching Total Revenue:**
   * Uses the **useEffect** hook to initiate the total revenue fetching process when the component mounts.
   * Checks if the user is logged in; if not, displays an error toast and redirects to the login page.
   * Defines the **totalRevenue** function to make a GET request to the totalRevenue endpoint, providing the user's ID.
   * Sets the total revenue in the component's state.
5. **Rendering:**
   * Returns JSX to render the component.
   * Includes the **Navbar** component.
   * Displays the total revenue with appropriate styling.

**TopSales.js**

1. **Component Structure:**
   * Imports necessary React components and hooks (e.g., **useEffect**, **useState**) from React and third-party libraries.
   * Imports components and functions needed for this specific component, such as **Navbar** and **toast** for notifications.
   * Utilizes Redux hooks (**useSelector**) to access user data from the Redux state.
   * Utilizes React Router's **useNavigate** hook for navigation.
2. **State and Variables:**
   * Defines a state variable **top5Sales** using the **useState** hook to store the top 5 sales data.
3. **Request Configuration:**
   * Defines a request configuration (**reqConfig**) with an Authorization header containing the user's token.
4. **Fetching Top 5 Sales:**
   * Uses the **useEffect** hook to initiate the top 5 sales fetching process when the component mounts.
   * Checks if the user is logged in; if not, displays an error toast and redirects to the login page.
   * Defines the **topSales** function to make a GET request to the topSales endpoint, providing the user's ID.
   * Sets the top 5 sales data in the component's state.
5. **Rendering:**
   * Returns JSX to render the component.
   * Includes the **Navbar** component.
   * Displays the top 5 sales in a table with columns for sales ID, product name, quantity, and sale amount.

**Register.js**

* **Imports:**
  + React, **useEffect** and **useState** from React library.
  + **Navbar** component for rendering the navigation bar.
  + **axios** for making HTTP requests.
  + **toast** for displaying notifications.
  + **REACT\_APP\_API\_USER** constant for the API endpoint related to user operations.
  + **useNavigate** from 'react-router-dom' for programmatic navigation.
* **Component Definition:**
  + **Register** is a functional component.
  + Initializes state variables to manage form input values and loading status.
  + Retrieves the **navigate** function from React Router for navigation.
* **Lifecycle:**
  + Uses **useEffect** to check if the user is already logged in. If so, it displays a success message and redirects to the 'addSales' page.
* **Functionality:**
  + **register** function handles the user registration process:
    - It prevents the default form submission.
    - Sets the loading state to true.
    - Constructs the request data object with user input.
    - Makes a POST request to the user registration endpoint.
    - Handles the response based on the HTTP status:
      * If successful (status 201), displays a success message, resets the form, and redirects to the login page.
      * If unsuccessful, displays an error message and redirects to the login page.
* **Render:**
  + Renders the **Navbar** component.
  + Displays a registration form with input fields for first name, last name, email, and password.
  + Includes a submit button that triggers the registration process.
  + Uses conditional rendering for a loading spinner when the form is being submitted.

**Login.js**

* **Imports:**
  + React, **useEffect**, and **useState** from React library.
  + **Navbar** component for rendering the navigation bar.
  + **useNavigate** from 'react-router-dom' for programmatic navigation.
  + **axios** for making HTTP requests.
  + **REACT\_APP\_API\_USER** constant for the API endpoint related to user operations.
  + **toast** for displaying notifications.
  + **useDispatch** from 'react-redux' for dispatching actions to the Redux store.
* **Component Definition:**
  + **Login** is a functional component.
  + Initializes state variables to manage email, password, and loading status.
  + Retrieves the **navigate** function from React Router for navigation.
  + Retrieves the **dispatch** function from React Redux for dispatching actions.
* **Lifecycle:**
  + Uses **useEffect** to check if the user is already logged in. If so, it displays a success message and redirects to the 'addSales' page.
* **Functionality:**
  + **login** function handles the user login process:
    - It prevents the default form submission.
    - Sets the loading state to true.
    - Constructs the request data object with user email and password.
    - Makes a POST request to the user login endpoint.
    - Handles the response based on the HTTP status:
      * If successful (status 201), displays a success message, stores the token and user data in local storage, dispatches a Redux action for login success, and redirects to the 'addSales' page.
      * If unsuccessful, displays an error message.
* **Render:**
  + Renders the **Navbar** component.
  + Displays a login form with input fields for email and password.
  + Includes a submit button that triggers the login process.
  + Uses conditional rendering for a loading spinner when the form is being submitted.

**HomePage.js**

* **Imports:**
  + React for creating React components.
  + **Navbar** component for rendering the navigation bar.
* **Component Definition:**
  + **Homepage** is a functional component.
* **Render:**
  + Renders the **Navbar** component.
  + Displays content within a **<div>** container.
  + Includes a page title with a welcome message.
  + Includes a subtitle with instructions for the user, encouraging them to select tasks from the navigation bar based on their needs.
* **Styling:**
  + Applies styling to center the content using the **textAlign** property.
  + Adjusts the margin-top of the title and subtitle for spacing.

**AddSales.js**

* **Imports:**
  + React for creating React components.
  + **Navbar** component for rendering the navigation bar.
  + **useSelector** from react-redux for accessing data from the Redux store.
  + **toast** from react-toastify for displaying notifications.
  + **axios** for making HTTP requests.
  + **REACT\_APP\_API\_SALE** from the configuration for the API endpoint.
  + **useNavigate** from react-router-dom for programmatic navigation.
* **Component Definition:**
  + **AddSales** is a functional component.
* **State:**
  + State variables (**productName**, **quantity**, **amount**, **loading**) to manage form input values and loading state.
* **Redux:**
  + Uses the **useSelector** hook to get user data from the Redux store.
* **Effects:**
  + Uses **useEffect** to check if the user is logged in. If not, redirects to the login page.
* **Form Submission:**
  + **addSales** function handles the form submission.
  + Creates a request data object with product name, quantity, amount, and author (user ID).
  + Sets up a request configuration with an authorization header.
  + Makes a POST request to the **addSales** endpoint.
  + Displays success or error messages based on the response status.
  + Resets the form fields after submission.
* **Render:**
  + Renders the **Navbar** component.
  + Displays a form with input fields for product name, quantity, and amount.
  + Includes a submit button that shows a loading spinner during form submission.
* **Styling:**
  + Applies styling to center the content and style the form container.

**Components**

**Navbar.js**

* **Imports:**
  + React for creating React components.
  + **NavLink** and **useNavigate** from react-router-dom for navigation.
  + **useDispatch** from react-redux for dispatching actions to the Redux store.
  + **toast** from react-toastify for displaying notifications.
  + A separate CSS file (**Navbar.css**) for styling.
* **Component Definition:**
  + **Navbar** is a functional component.
* **Hooks:**
  + Uses **useDispatch** to get the dispatch function for Redux actions.
  + Uses **useNavigate** to get the navigate function for programmatic navigation.
* **Checking User Data:**
  + Checks if user data is present in local storage (**localUser** variable).
* **Logout Function:**
  + Defines a **logout** function to handle the logout process.
  + Removes the token and user data from local storage.
  + Displays a success toast on logout.
  + Dispatches a Redux action (**LOGIN\_ERROR**) to update the state.
  + Navigates to the login page.
* **Render:**
  + Renders a navigation bar with a red background.
  + Includes a brand link to the homepage.
  + Utilizes a responsive design with a toggler button for smaller screens.
  + Displays navigation links for adding sales, top sales, total revenue, login, register, and logout (if the user is logged in).
* **Styling:**
  + Assumes the presence of a separate CSS file (**Navbar.css**) for styling.

**Navbar.css**

* **Selector:**
  + Targets elements with the class **.nav-link.active**.
* **Styling:**
  + Sets the text color to white (**color: white**).
  + Uses **!important** to prioritize this styling even if there are conflicting styles elsewhere.
* **Purpose:**
  + Typically, this styling is used to visually distinguish the active navigation link by changing its text color.

**Screenshots**









