Understanding the Project Structure

Based on the files, the project has a Node.js backend folder for a currency calculator application. Here's a breakdown of the project structure:

- **server.js:** This is the main entry point of the application. It sets up the Express server, middleware, and routes.
- **routes/:** This directory contains the route definitions for different parts of your application (user authentication and currency management).
- **utils/:** This directory (which you named as utils) contains the controller logic, database connection, and middleware.
- **middleware/:** Holds middleware functions, such as authentication middleware.

Backend File-by-File Analysis

1. server.js

- **Functionality:** This file sets up the Express server, the core of the Node.js application. It handles incoming requests and sends responses.
- **Logic:**
  - It imports necessary libraries: `express` for the server, `body-parser` for handling request bodies, `cors` for cross-origin requests, route files (`currencyRoutes`, `userRoutes`), authentication middleware (`authenticate`), and `dotenv` for environment variables.
  - It creates an Express application instance (`app`).
  - It configures middleware:
    - `cors()` enables Cross-Origin Resource Sharing, allowing requests from different domains.
    - `bodyParser.json()` parses JSON request bodies.
    - `bodyParser.urlencoded({ extended: true })` parses URL-encoded request bodies.
  - It defines routes:
    - `/api/currencies` is handled by `currencyRoutes` (for currency-related operations).
    - `/api/users` is handled by `userRoutes` (for user authentication).
    - `/` responds with a welcome message.
  - It starts the server and listens on the specified port.

2. userRoutes.js

Functionality: This file defines the routes for user-related operations (signup and login).

- **Logic:**
  - It imports `express` and the `userController` (which contains the logic for user operations).
  - It creates an Express Router instance.
  - It defines two routes:
    - `/signup` (POST): Handles user registration, calling the `signupUser` function in the `userController`.

        - `/login` (POST): Handles user login, calling the `loginUser` function in the `userController`.
    - It exports the router so it can be used by the main application (server.js).

3. currencyRoutes.js

Functionality: This file defines the routes for currency-related operations (CRUD and conversion).

- **Logic:**
    - It imports `express`, the `currencyController`, and the `authenticate` middleware.
    - It creates an Express Router.
    - It defines the following routes, all of which are protected by the `authenticate` middleware:
        - `/` (GET): Retrieves all currencies (`getAllCurrencies`).
        - `/:code` (GET): Retrieves a specific currency by its code (`getCurrency`).
        - `/` (POST): Creates a new currency (`createCurrency`).
        - `/:code` (PUT): Updates a currency (`updateCurrency`).
        - `/:code` (DELETE): Deletes a currency (`deleteCurrency`).
        - `/convert/:from/:to/:amount` (GET): Converts currency (`convertCurrency`).
    - It exports the router.

4. userController.js

Functionality: This file contains the logic for user signup and login.

- **Logic:**
    - It imports the database connection pool (`pool`), `jsonwebtoken` for generating tokens, and `bcrypt` for password hashing.
    - `signupUser`:
        - It extracts the `username` and `password` from the request body.
        - It validates that both are provided.
        - It checks if the username already exists in the database.
        - It hashes the password using `bcrypt`.
        - It inserts the new user into the database.
        - It sends a success response with status 201.
        - It handles errors and sends an error response with status 500.
    - `loginUser`:
        - It extracts `username` and `password` from the request.
        - It retrieves the user from the database by username.
        - It checks if the user exists.
        - It compares the provided password with the hashed password from the database using `bcrypt.compare()`.
        - If the passwords match, it generates a JWT token and sends it in the response.
        - It handles invalid credentials and other errors, sending appropriate error responses.

5. currencyController.js

Functionality: This file contains the logic for handling currency-related operations: getting currencies, getting a single currency, creating, updating, and deleting currencies, and converting between currencies.

- **Logic:** Each function interacts with the database using the `pool` to perform queries. They also handle errors and send appropriate responses to the client. Note the consistent pattern of including the `user_id` in database queries, ensuring data isolation for each user.
    - `getAllCurrencies`: Retrieves all currencies for the logged-in user.
    - `getCurrency`: Retrieves a specific currency by code for the logged-in user.
    - `createCurrency`:
        - Extracts currency data from the request body.
        - Validates that all required data is present.
        - Checks if a currency with the given code already exists for the user.
        - Inserts the new currency into the database.
        - Sends a success response.
    - `updateCurrency`:
        - Extracts the currency code from the request parameters and the new data from the request body.
        - Checks if the currency exists for the user.
        - Updates the currency in the database.
        - Retrieves the updated currency and sends it in the response.
    - `deleteCurrency`:
        - Extracts the currency code from the request parameters.
        - Checks if the currency exists for the user.
        - Deletes the currency from the database.
        - Sends a success response (204 No Content).
    - `convertCurrency`:
        - Extracts the `from`, `to`, and `amount` from the request parameters.
        - Retrieves the exchange rates for the `from` and `to` currencies for the user.
        - Validates that both currencies exist.
        - Performs the currency conversion.
        - Sends the converted amount in the response.

6. database.js

*** **Functionality:** This file establishes the connection to your MySQL database using the \`mysql2/promise\` library. ***

****Logic:****

* It imports the \`mysql2/promise\` library for asynchronous database operations and \`dotenv\` to load environment variables.

* It creates a connection pool using \`mysql.createPool()\`. A connection pool is a set of database connections that can be reused, which improves performance.

* The connection details (host, user, password, database name, port) are retrieved from environment variables using `process.env`.This is a good practice for security and configuration. It also provides a default port value of 3306 if `DB_PORT` is not defined.

* `connectionLimit` sets the maximum number of connections in the pool.

* It exports the connection pool (`pool`) so it can be used by other parts of the application.

****7. authMiddleware.js****

- **Functionality:** This file contains the `authenticate` middleware, which is used to protect routes. It verifies the JWT token sent in the `Authorization` header of requests.
- **Logic:**

- It imports `jsonwebtoken` for verifying tokens and the database connection pool (`pool`).
- `authenticate`:
    - It retrieves the token from the `Authorization` header, removing the "Bearer " prefix if present.
    - If no token is provided, it sends a 401 Unauthorized error.
    - It verifies the token using `jwt.verify()` with the secret key (`process.env.SECRET_KEY`).
    - It decodes the token and retrieves the user from the database based on the username in the token's payload.
    - If no user is found, it sends a 401 Unauthorized error (invalid token).
    - It attaches the user object to the request (`req.user`) so that subsequent route handlers can access user information. This is how the `user_id` is obtained in the controller functions.
    - It calls `next()` to pass control to the next middleware or route handler.
    - It handles errors during token verification or database lookup and sends a 401 Unauthorized error.

Frontend File-by-File Analysis

Here's a breakdown of the React frontend code:

1. App.js

- **Functionality:** This is the main component of your React application. It sets up the routing, manages user authentication state, and renders the navigation bar.
- **Logic:**
    - It imports necessary React libraries, routing components from `react-router-dom`, components for different pages (`CurrencyConverter`, `CurrencyForm`, `Login`, `Signup`, `HomePage`), and Bootstrap components for styling.
    - It uses the `useState` hook to manage the `isLoggedIn` (boolean) and `token` (string) states.
    - `useEffect` is used to check for a stored token in `localStorage` when the app loads. If a token is found, it sets the `token` and `isLoggedIn` states, persisting the user's session.

    - `handleLogin` is called after successful login. It updates the `token` and `isLoggedIn` state and stores the token in `localStorage`. It then redirects the user to the home page (`/`). **Important:** It uses `window.location.href = '/';` for navigation. While this works, using `navigate('/')` from `react-router-dom` is generally preferred within React Router for smoother transitions (though you have it commented out).
    - `handleLogout` clears the token from the state, sets `isLoggedIn` to `false`, removes the token from `localStorage`, and redirects to the home page. Again, it uses `window.location.href`.
    - The `Router`, `Routes`, and `Route` components from `react-router-dom` define the application's navigation structure.
    - A Bootstrap `Navbar` is used for the navigation menu. It conditionally renders links based on the `isLoggedIn` state.
    - The `Routes` component renders the appropriate component based on the current URL path. Routes `/form` and `/CurrencyConverter` are protected, only rendering their respective components if the user is logged in.

2. index.js

- **Functionality:** This is the entry point for your React application. It renders the `App` component into the `root` element of your HTML.
- **Logic:**
    - It imports `React` and `ReactDOM`.
    - It gets the DOM element with the ID `root` (this element should be in your `index.html` file).
    - It uses `ReactDOM.createRoot` to create a root and then renders the `App` component inside it. The `<React.StrictMode>` component enables extra checks and warnings for potential problems in your application (recommended for development).

3. Signup.js

- **Functionality:** This component handles user registration (signup).
- **Logic:**
    - It imports React hooks, `axios` for making HTTP requests, Bootstrap components for styling, and `Link` from `react-router-dom` for navigation.
    - It uses `useState` to manage the form fields (`username`, `password`, `confirmPassword`), error messages (`error`), and loading state (`loading`).
    - `handleSubmit` is called when the signup form is submitted.
        - It prevents the default form submission behavior.
        - It sets `loading` to `true` and clears any previous errors.
        - It checks if the `password` and `confirmPassword` match. If not, it sets an error and returns.
        - It uses `axios.post` to send a POST request to the `/api/users/signup` endpoint with the username and password.

        - If the signup is successful, it redirects the user to the login page using `window.location.href = '/login';` (again, consider using `navigate('/login')`).
        - If there's an error during signup, it extracts the error message from the backend's response (if available) or displays a generic error message.
        - Finally, it sets `loading` to `false`.
    - The `return` statement renders the signup form using Bootstrap components. It includes input fields for username, password, and confirm password, an error message display, and a link to the login page.

4. Login.js

- **Functionality:** This component handles user login.
- **Logic:**
    - It's very similar to `Signup.js`. It imports the same libraries and uses `useState` to manage form fields, errors, and loading state.
    - The key difference is in `handleSubmit`:
        - It sends a POST request to `/api/users/login`.
        - If the login is successful, it extracts the `token` from the response data.
        - It calls the `onLogin` function passed down from `App.js` (this function updates the token and login status in the main `App` component).
        - It redirects the user to the home page using `window.location.href = '/';` (again, consider using `navigate('/')`).
    - The `return` statement renders the login form, which is simpler than the signup form (no confirm password field).

5. CurrencyConverter.js

* **Functionality:** This component allows users to convert currencies. *

Logic:
* It imports React hooks, `axios`, and Bootstrap components.
* It uses `useState` to manage the list of currencies (`currencies`), the selected "from" and "to" currencies (`fromCurrency`, `toCurrency`), the amount to convert (`amount`), the converted amount (`convertedAmount`), error messages (`error`), and loading state (`loading`).
* **`fetchCurrencies`:**
* It fetches the list of currencies from the `/api/currencies` endpoint using `axios.get`.
* It includes the `Authorization` header with the JWT token to authenticate the request.
* It updates the `currencies` state with the fetched data.
* It handles errors and sets the `error` state if the request fails.
* **`useEffect`:**
* It calls `fetchCurrencies` when the `token` prop changes (i.e., when the user logs in). This ensures that the list of currencies is fetched after successful authentication. * `handleConvert`:
* **`handleConvert`:**
* It's called when the user clicks the "Convert" button.
* It validates that `fromCurrency`, `toCurrency`, and `amount` are provided.
* It sends a GET request to the `/api/currencies/convert/:from/:to/:amount` endpoint with the selected currencies and amount.
* It updates the `convertedAmount` state with the result from the server.
* It handles errors and sets the `error` state if the conversion fails.

* The `return` statement renders the currency converter form.
* It uses Bootstrap's `Form` and `Form.Control` components for the UI.
* It uses `<select>` elements to allow the user to choose the "from" and "to" currencies. The options are populated from the `currencies` state.
* It displays the converted amount in an `Alert` component.

6. CurrencyForm.js

Functionality: This component allows users to perform CRUD (Create, Read, Update, Delete) operations on currencies. This is likely where you're encountering the "failed to add currency" issue.

- **Logic:**
  - It imports React hooks, `axios`, and Bootstrap components (including `Table`).
  - It uses `useState` to manage:
    - `currencies`: The list of currencies.
    - `code`, `name`, `rate`: The form fields for creating/updating currencies.
    - `error`, `successMessage`: For displaying feedback to the user.
    - `loading`: To show a loading state.
    - `editingCode`: The code of the currency being edited (null if creating).
  - `fetchCurrencies`:
    - Fetches the list of currencies, similar to `CurrencyConverter.js`.
  - `useEffect`:
    - Calls `fetchCurrencies` when the `token` changes.
  - `handleCreate`:
    - Called when the form is submitted in "create" mode (i.e., when `editingCode` is null).
    - It sends a POST request to `/api/currencies` with the currency data.
    - It clears the form fields and displays a success message.
    - It calls `fetchCurrencies` to refresh the currency list.
    - It handles errors and displays an error message.
  - `handleUpdate`:
    - Called when the form is submitted in "edit" mode (i.e., when `editingCode` is not null).
    - It sends a PUT request to `/api/currencies/:code` to update the currency.
    - It clears the form, resets `editingCode`, and displays a success message.
    - It calls `fetchCurrencies` to update the list.
    - It handles errors.
  - `handleDelete`:
    - Called when the user clicks the "Delete" button.
    - It shows a confirmation dialog before deleting.
    - It sends a DELETE request to `/api/currencies/:code`.
    - It displays a success message and refreshes the currency list.
    - It handles errors.

- `handleEdit`:
    - Called when the user clicks the "Edit" button.
    - It populates the form with the currency data to be edited.
- `handleCancelEdit`:
    - Called when the user clicks the "Cancel Edit" button.
    - It clears the form and resets `editingCode`.
- The `return` statement renders:
    - A form for creating/updating currencies. The form changes slightly depending on whether you are editing or creating.
    - A table displaying the list of currencies.
    - Buttons for editing and deleting currencies.

7. HomePage.js

Functionality: This is a simple homepage that displays a welcome message and indicates whether the user is logged in.

- **Logic:**

- It receives the `isLoggedIn` prop from `App.js`.
- It conditionally renders a message based on the `isLoggedIn` value.