**ADDITIONAL HANDSON SOLUTIONS-WEEK-07**

***-AAISHA SULTANA GUDURU 6384619***

***13-REACTJS-HOL(already done in mandatory handson)***

**EXERCISE-01 14 -REACTJS-HOL**

**IMPLEMENTATION:**

**1. Project Setup and Initial Exploration:**

The provided starter application was restored by running npm install in the project directory, followed by npm start to launch it in the browser. The existing application structure was examined, focusing on App.js, EmployeesList.js, and EmployeeCard.js, to understand how the theme value was being propagated via props through multiple component levels.

**2. Creating the Context:**

A dedicated file, ThemeContext.js, was created in the src directory. The React createContext function was used to define a ThemeContext object with a default value of "light". This object was exported as the default export, ensuring that it could be easily imported into any component requiring theme access.

**3. Providing the Context Value:**

In App.js, ThemeContext was imported, and the application’s content was wrapped inside ThemeContext.Provider. A state variable theme was defined using the useState hook to manage the current theme value, along with a toggleTheme function to switch between "light" and "dark". The value prop of the Provider was set to theme, making it available to all descendant components. Importantly, the theme prop was removed from the EmployeesList component invocation, as context now supplies it.

**4. Removing Prop Drilling in the Employee List:**

The EmployeesList.js component was updated to no longer receive or pass the theme prop. It now focuses solely on rendering the list of employee data, delegating theme styling responsibility to each EmployeeCard.

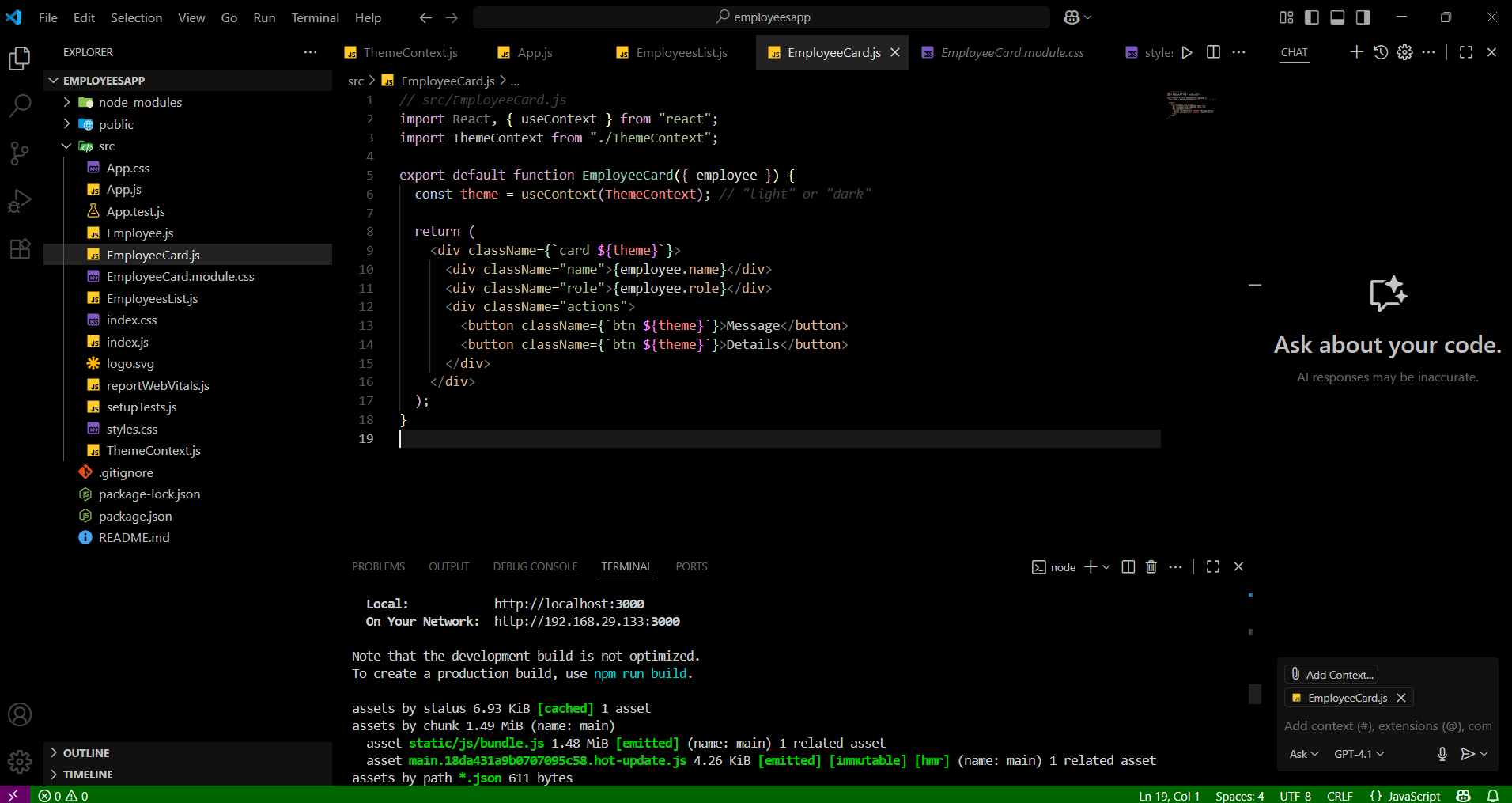
**5. Consuming Context in the Employee Card:**

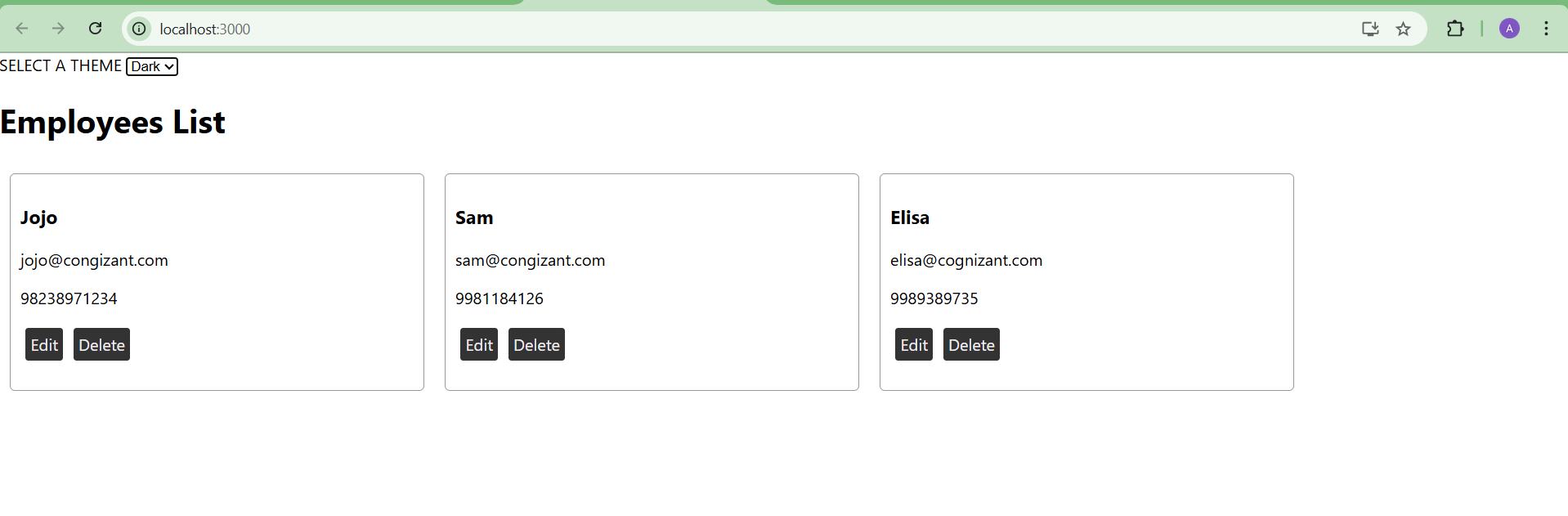
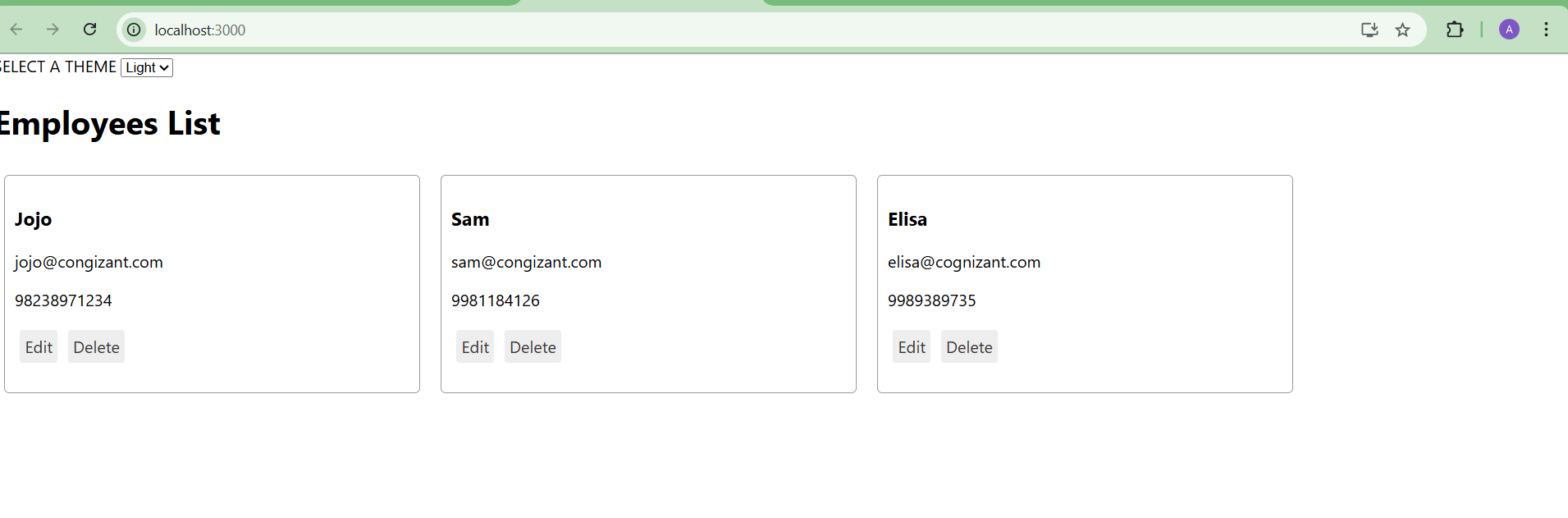
In EmployeeCard.js, the useContext hook was imported from React, along with the ThemeContext object. The theme value was retrieved directly from context and applied as a dynamic className to the enclosing card container and the action buttons. This removed the dependency on receiving theme via props, aligning with the Context API’s purpose.

To provide visual distinction between themes, CSS classes were defined for light and dark modes. These styles adjusted background colors, text colors, and button border colors accordingly. Applying these classes via context-driven values ensured that toggling the theme at the top level immediately updated the UI across all relevant components.

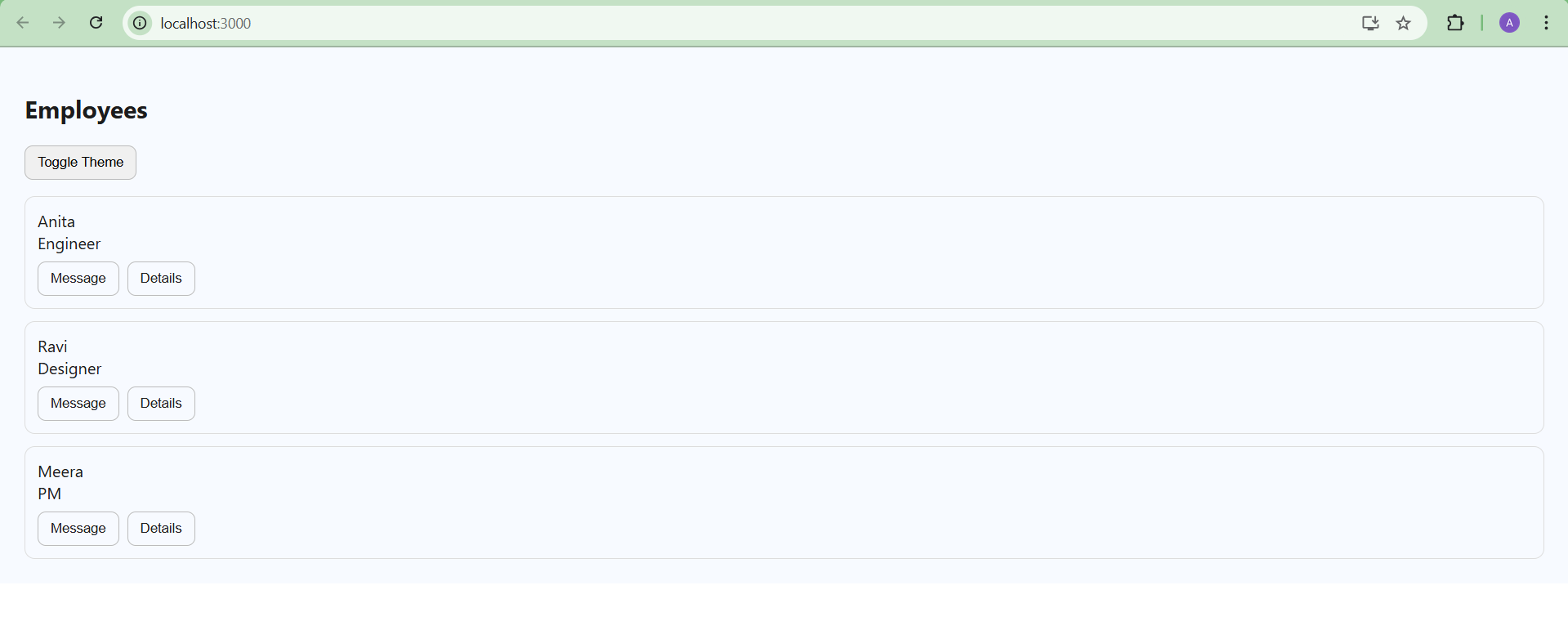
The application was run, and the Toggle Theme button was clicked to verify that all employee cards and their buttons correctly switched styles between light and dark themes. The removal of the theme prop from intermediate components confirmed that the Context API was functioning as intended, eliminating prop drilling while maintaining consistent styling throughout the component tree.

This implementation successfully transitioned the application from a prop-drilling architecture to a Context API-based state sharing approach. The theme value is now centrally managed and accessible by any nested component without manual prop forwarding.

****

**BEFORE-**

OUTPUT:





**EXERCISE-02 15-REACTJS-HOL**

**IMPLEMENTATION:**

The goal of this exercise was to design and implement a complaint registration form in React, enabling employees to raise complaints with their details and receive an auto-generated reference number for tracking purposes.

**1. Project Setup:**  
A new React application was created using create-react-app. The initial scaffold was cleaned by removing unused files, ensuring a minimal and organized starting point.

**2. Component Design:**  
A class-based component, ComplaintRegister, was created to encapsulate all form-related logic and state. The component maintained two state variables: employeeName and complaint. These were bound to form inputs as **controlled components**, ensuring the React state remained the single source of truth for form data.

**3. Form Structure**  
The form consisted of:

* A **textbox** for the employee’s name.
* A **textarea** for the complaint description.

Both fields were styled with padding, borders, and rounded corners for better readability.

**4. Submission Logic**  
The handleSubmit method was implemented to:

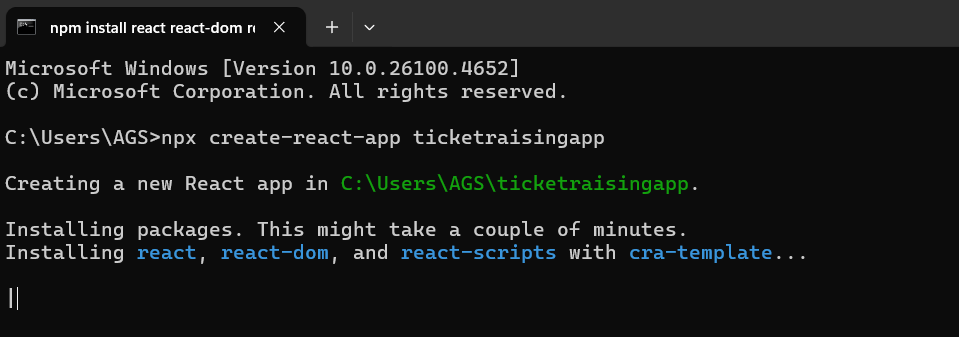
* Prevent the default browser form submission.
* Validate that both fields were filled.
* Generate a **unique Reference Number** using a combination of the current timestamp and a random six-digit number.
* Display an **alert** containing a success message, the generated reference number, and the entered details.

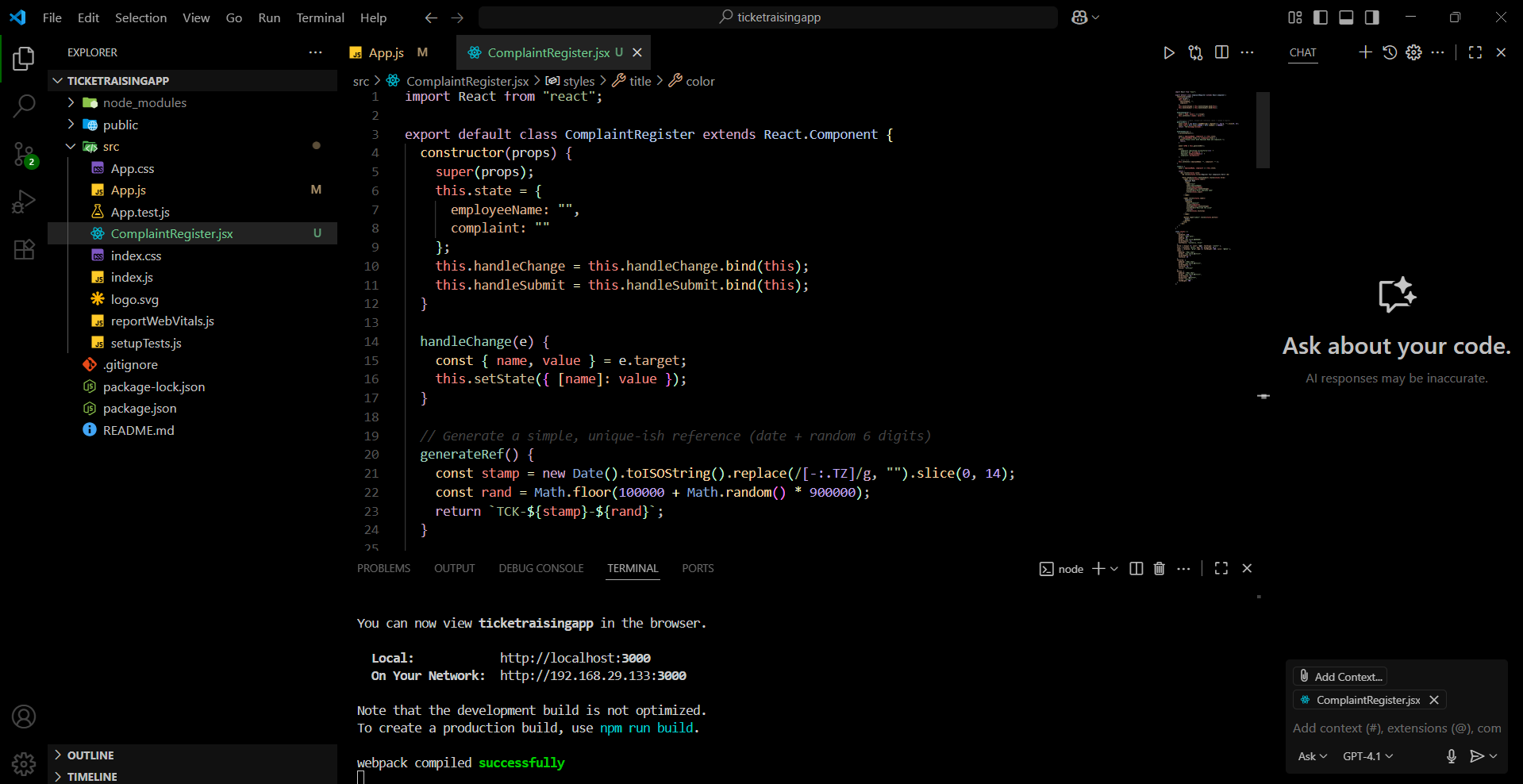
**5. Reference Number Generation**  
A helper function, generateRef(), constructed reference numbers in the format:  
TCK-YYYYMMDDHHMMSS-RANDOM6DIGITS.  
This ensured that each submission could be uniquely identified.

**6. State Reset**  
Upon successful submission, the component’s state was reset to clear the form fields, preparing the form for a new complaint without requiring a page reload.

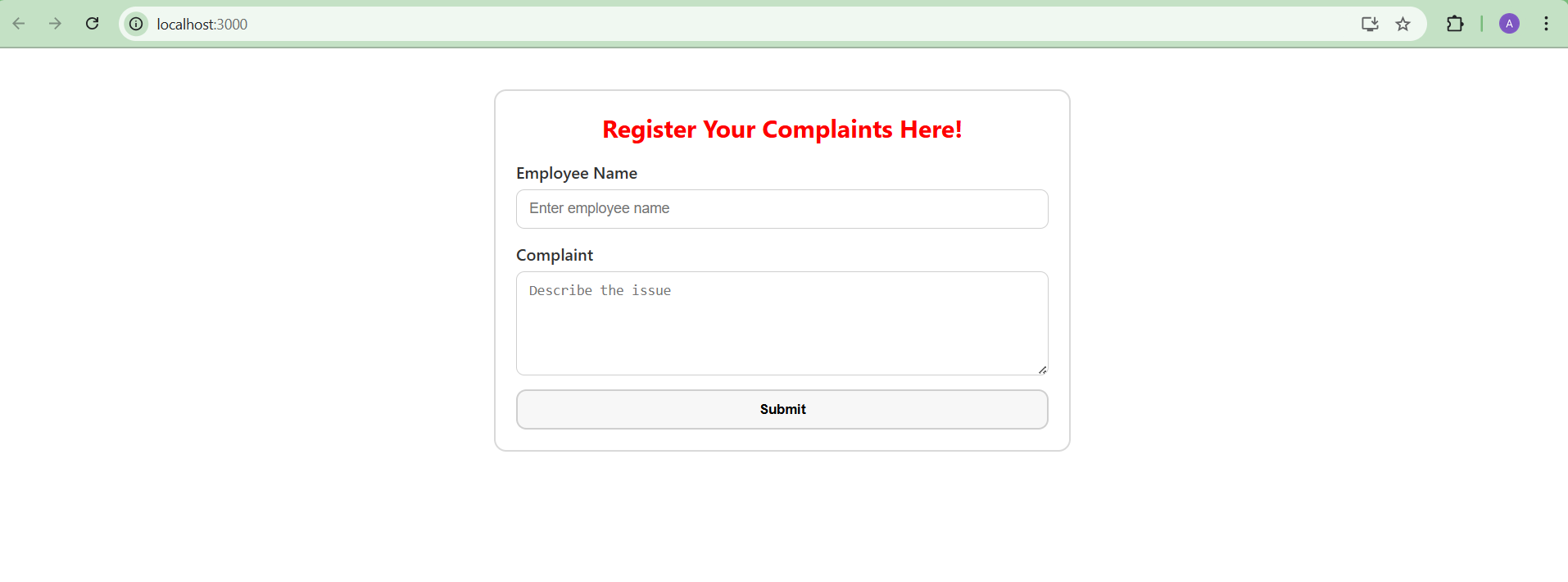
**7. Styling**  
The form and its elements were styled using an inline style object to provide a consistent card-like layout. This improved user experience while keeping styling simple and self-contained.

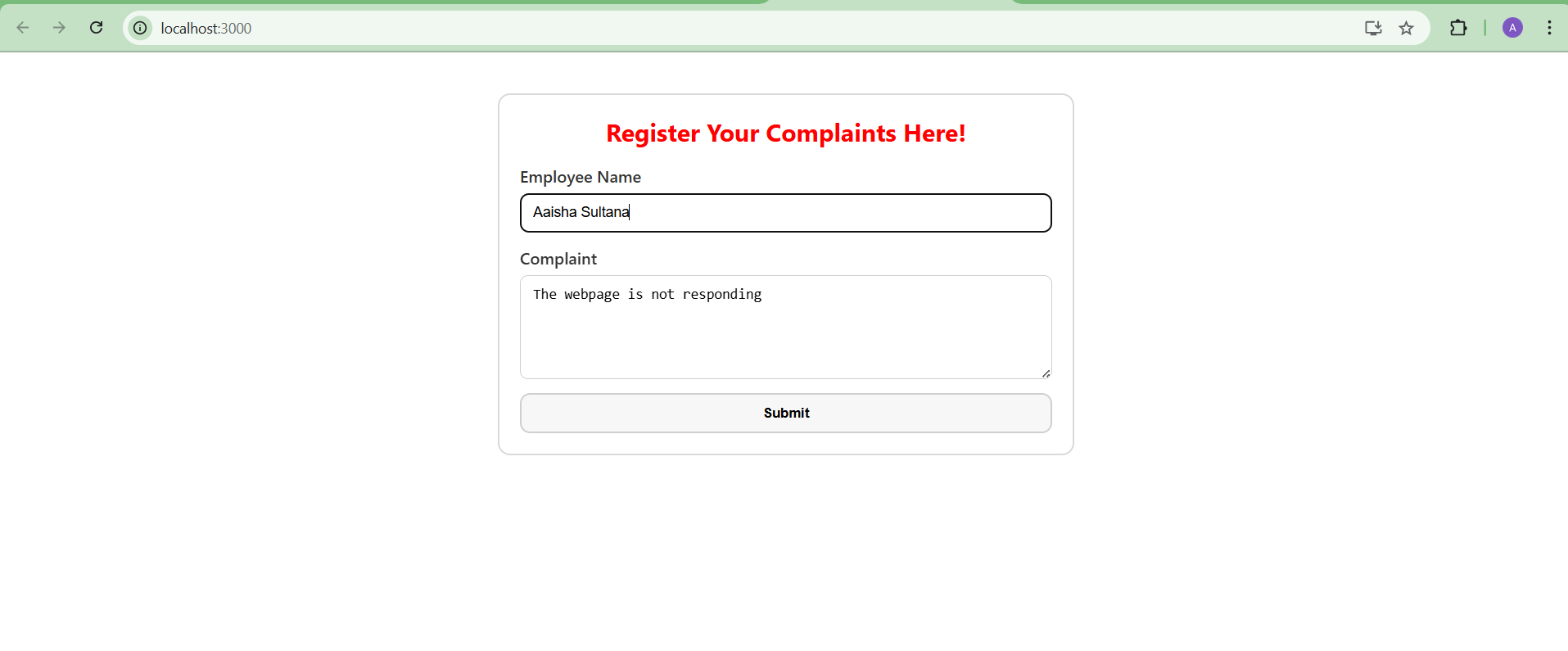
The ComplaintRegister component is imported into App.js and rendered as the main UI element of the application. This completes the integration, allowing the complaint form to be displayed when the application runs.

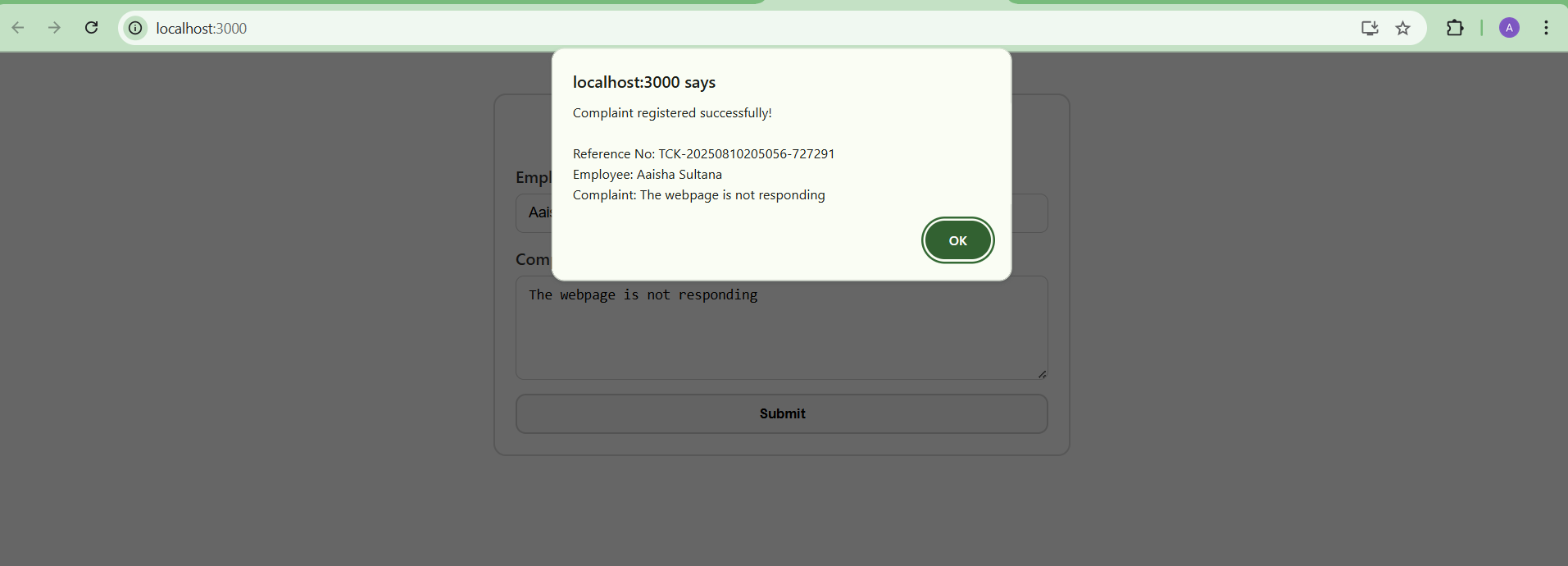
****

****

**OUTPUT:**

****

****

****

**EXERCISE-03 16-REACTJS-HOL**

**IMPLEMENTATION:**

**Architecture**

**Component:** Register (src/register.js) extends React.Component.

**State:** { name, email, password, errors } where errors holds per-field messages.

**Handlers:**

handleChange(e) — updates state and validates the changed field.

handleSubmit(e) — prevents default, validates all fields, alerts on success, resets state.

**Helper:** validateField(name, value) encapsulates rules.

**Validation Logic:**

* Name: minimum 5 characters.
* Email: must contain @ and . (basic syntactic check).
* Password: minimum 8 characters.
* On change: immediate feedback under each input.
* On submit: block submission if any error remains; otherwise alert success.

The Mail Register application implements a registration form in React using a class based component that encapsulates state, validation, and submission logic.

The component, defined in src/register.js, maintains name, email, password, and an errors map in its state to drive a controlled form pattern where UI values are always synchronized with component state.

Input handling is centralized in handleChange, which updates the relevant state key and immediately validates the changed field through a dedicated validateField(name, value) function.

This function enforces the required rules: a minimum length of five characters for the name, the presence of “@” and “.” for the email, and a minimum length of eight characters for the password.

Validation messages are stored per field and rendered inline beneath the corresponding input to provide instant, context-specific feedback.

Form submission is coordinated by handleSubmit, which first prevents the default browser action, then performs a full validation pass across all fields.

If any errors are present, submission is blocked and the user is prompted to correct the inputs.

When all fields are valid, the component displays a success alert summarizing the submitted name and email, and then resets the form state to clear the inputs for the next entry.

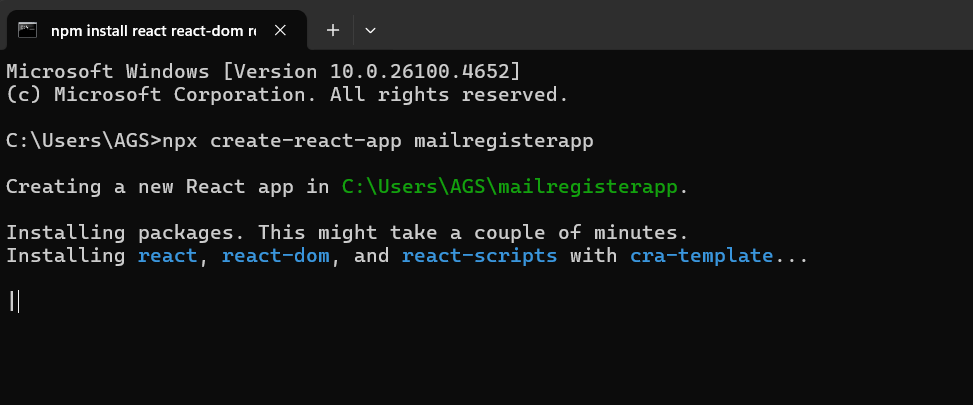
The layout prioritizes clarity and alignment. Each field row is rendered as a two-column CSS Grid with a fixed-width label column and a flexible input column, ensuring the label and its control sit side by side and line up across rows.

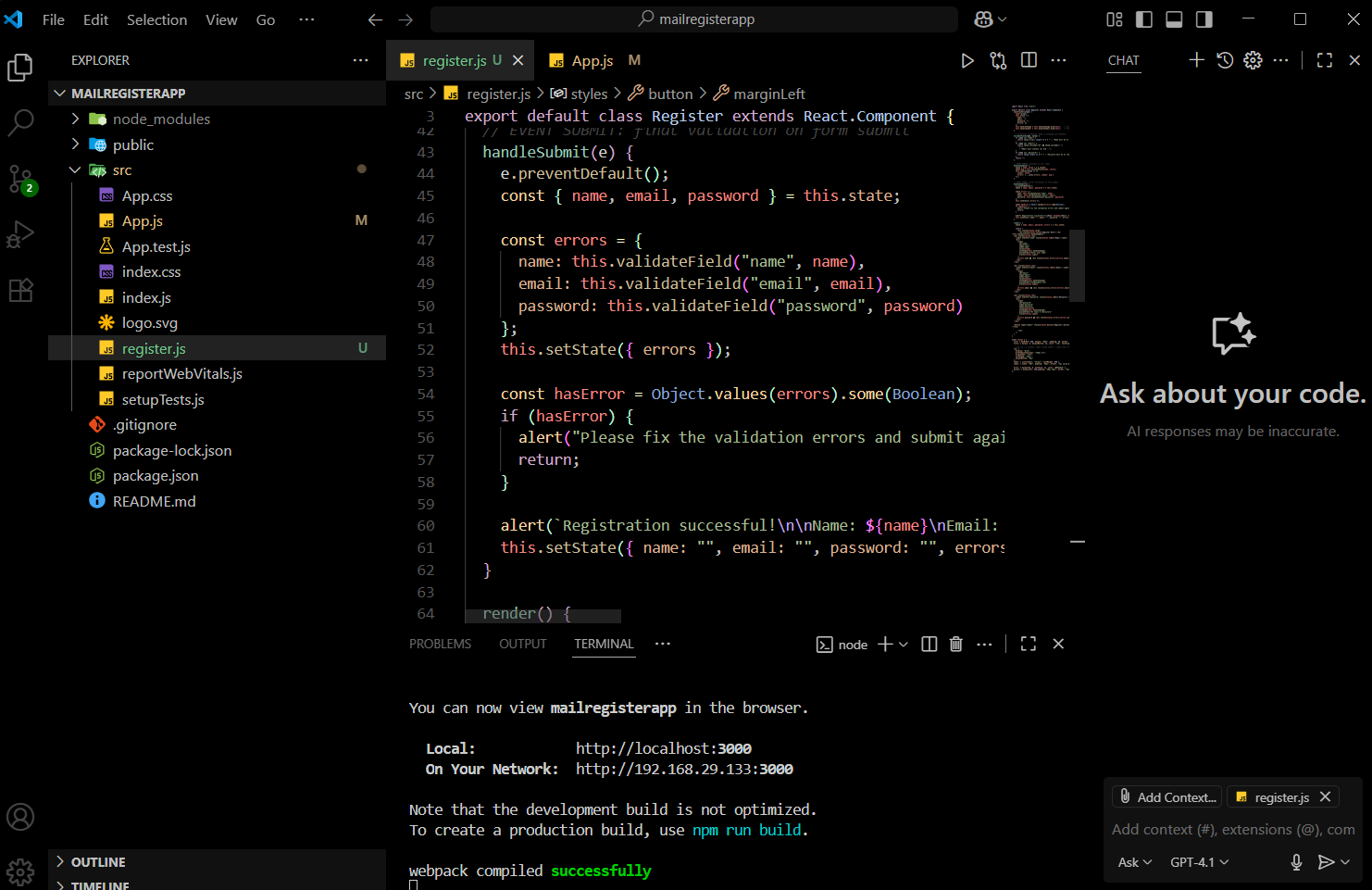
The submit button is placed centrally using either a flexbox wrapper with justify-content: center or by applying display: block with horizontal auto margins, resulting in a balanced and professional presentation.

Accessibility considerations include associating each <label> with its corresponding control via htmlFor/id, predictable tab order, and concise error messages positioned immediately after inputs for screen reader context.

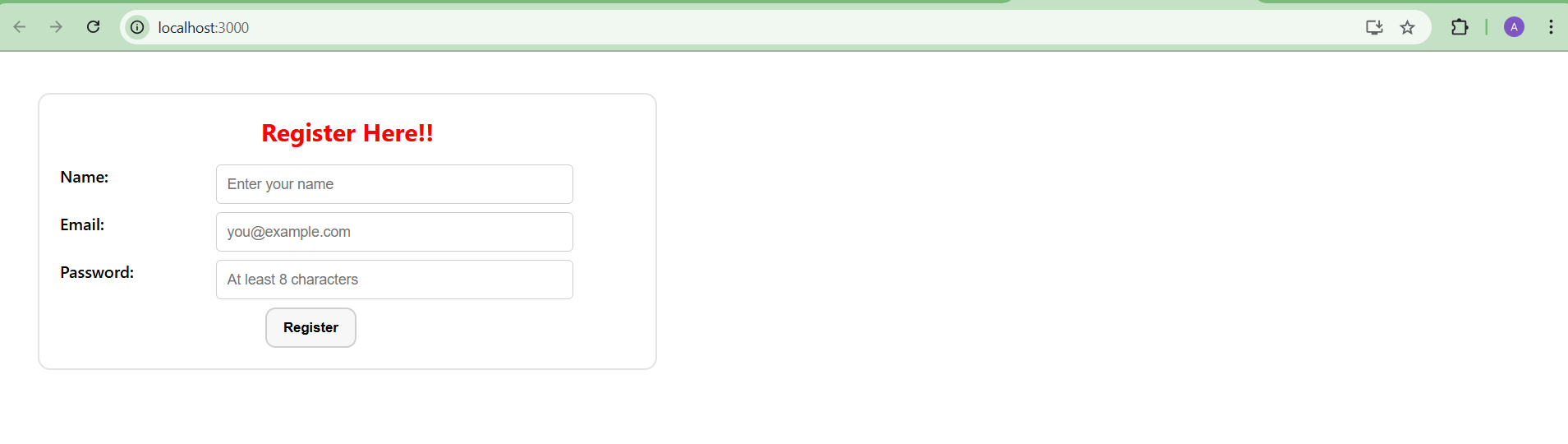
The solution is keyboard-friendly pressing Enter triggers form submission and the inline feedback reduces user friction during data entry.

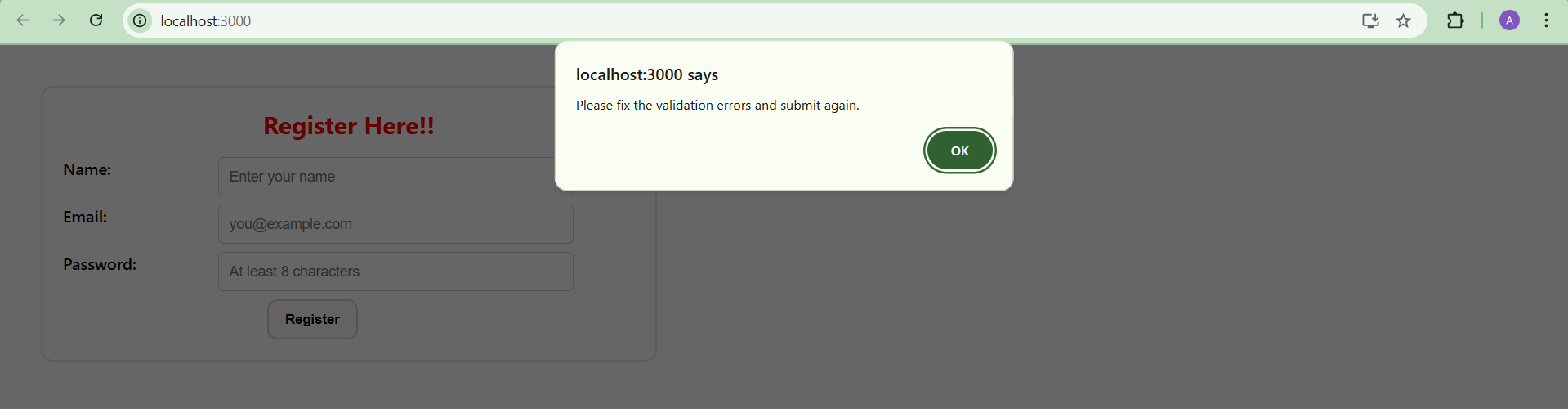
Overall, the implementation satisfies the functional and layout requirements with a clean class-component architecture, immediate and final validation flows, accessible markup, and a polished, side-by-side form layout with a centered submission control.

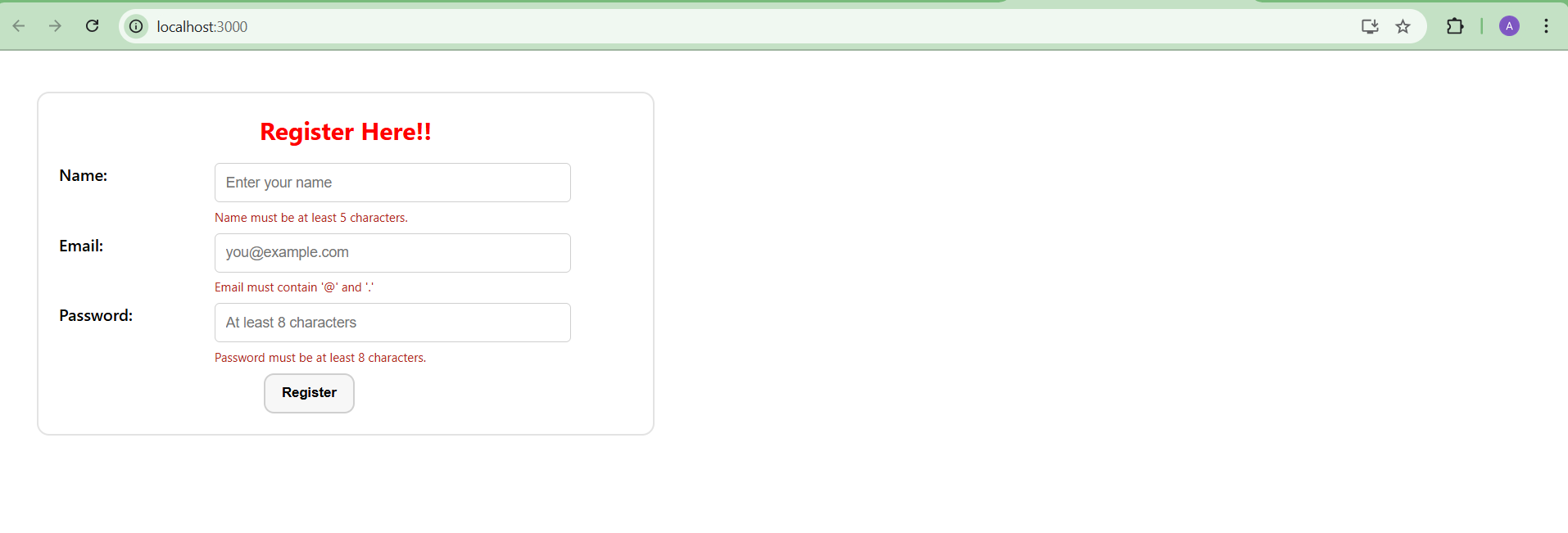
****

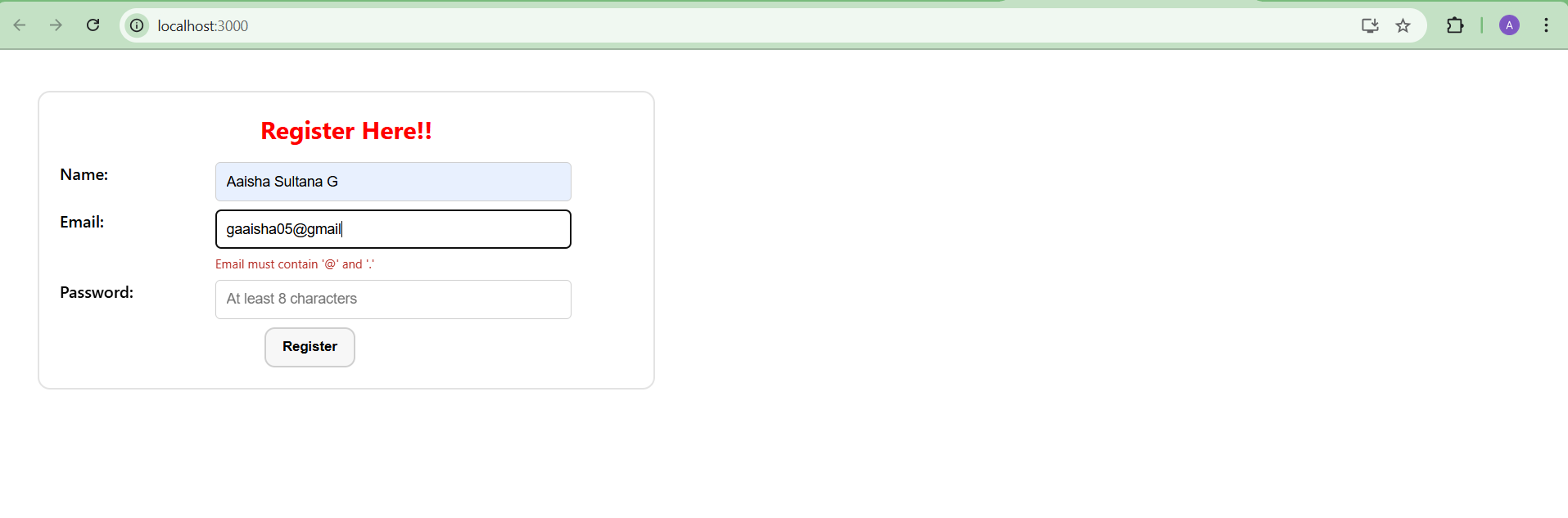
****

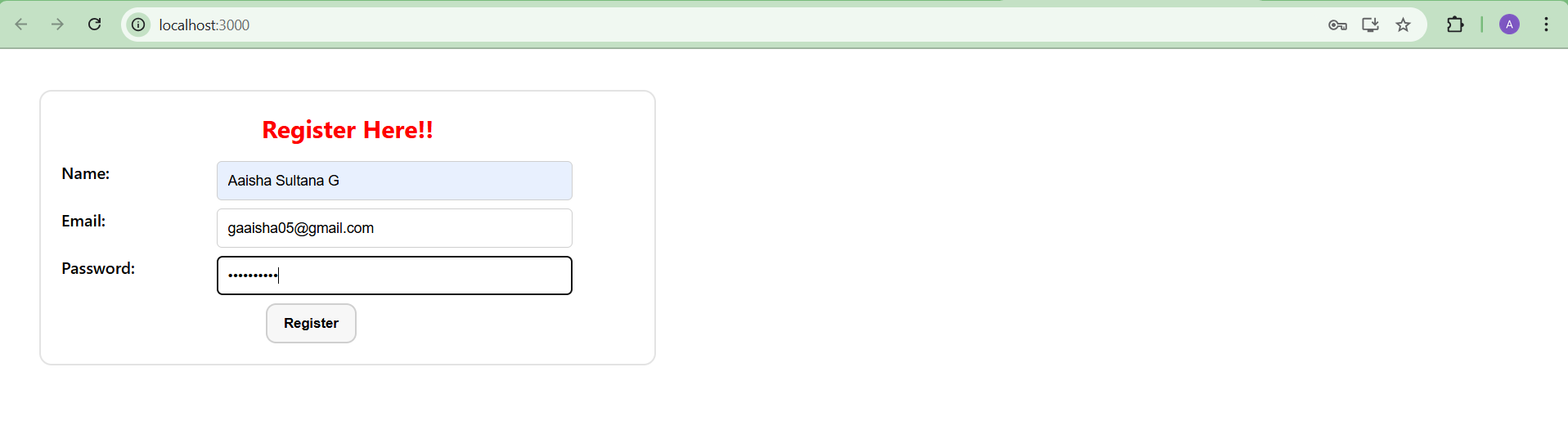
**OUTPUT:**

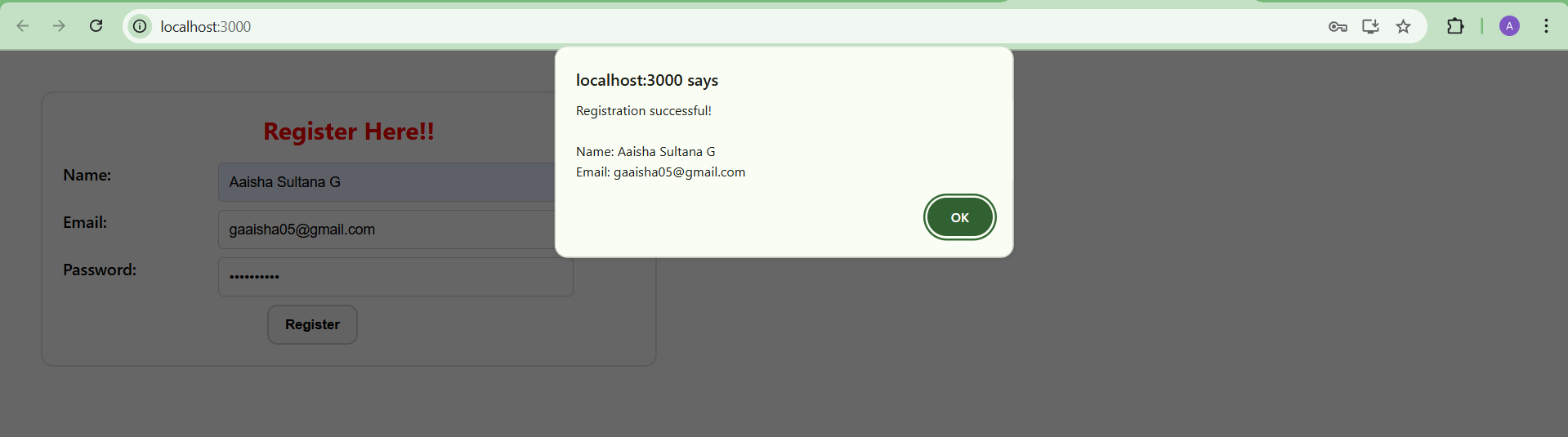
****

****

****

****

****

****

**EXERCISE-04 17-REACTJS-HOL**

**IMPLEMENTATION:**

The task is to build a React app that retrieves one record from https://api.randomuser.me/ and renders the user’s title, first name, last name, and photo in the exact layout shown in the reference: the full name appears as a bold, left aligned heading and the square image is displayed directly beneath it. The implementation must use a class component and perform the fetch in the mount lifecycle.

**Environment and Setup:**

A new project was created with create-react-app. The default scaffold was retained, and a single feature component named Getuser was added under src/. The app runs with npm start and opens at http://localhost:3000.

**Component Design and Data Flow:**

Getuser is a class component that initializes state to hold the fetched user object. The API call is issued in componentDidMount(). Once the response is parsed, the first result (results[0]) is stored in state and used for rendering. No intermediate props or context are required; all logic resides in the component.

**Rendering and Field Mapping:**

The render function reads the minimal fields required by the handson: name.title, name.first, name.last, and picture.large. These are concatenated into a single <h1>e.g., “Mr Donato Nunes”and the image is rendered with <img src={picture.large} />. No other attributes from the API are displayed.

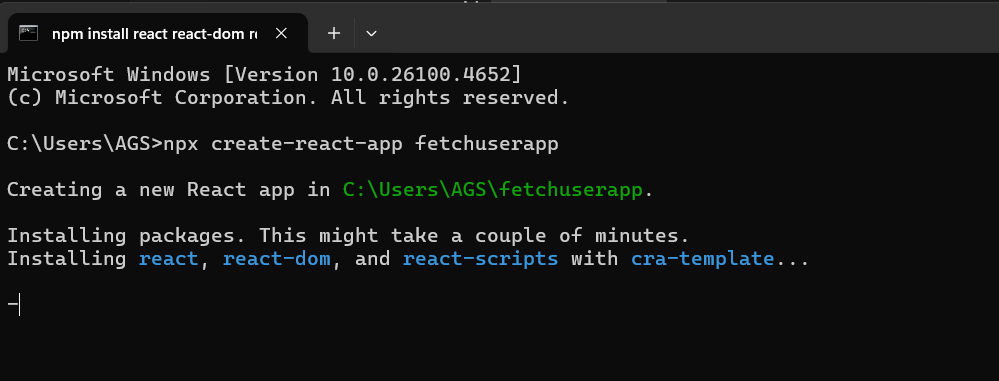
**Layout and Styling to Match Reference:**

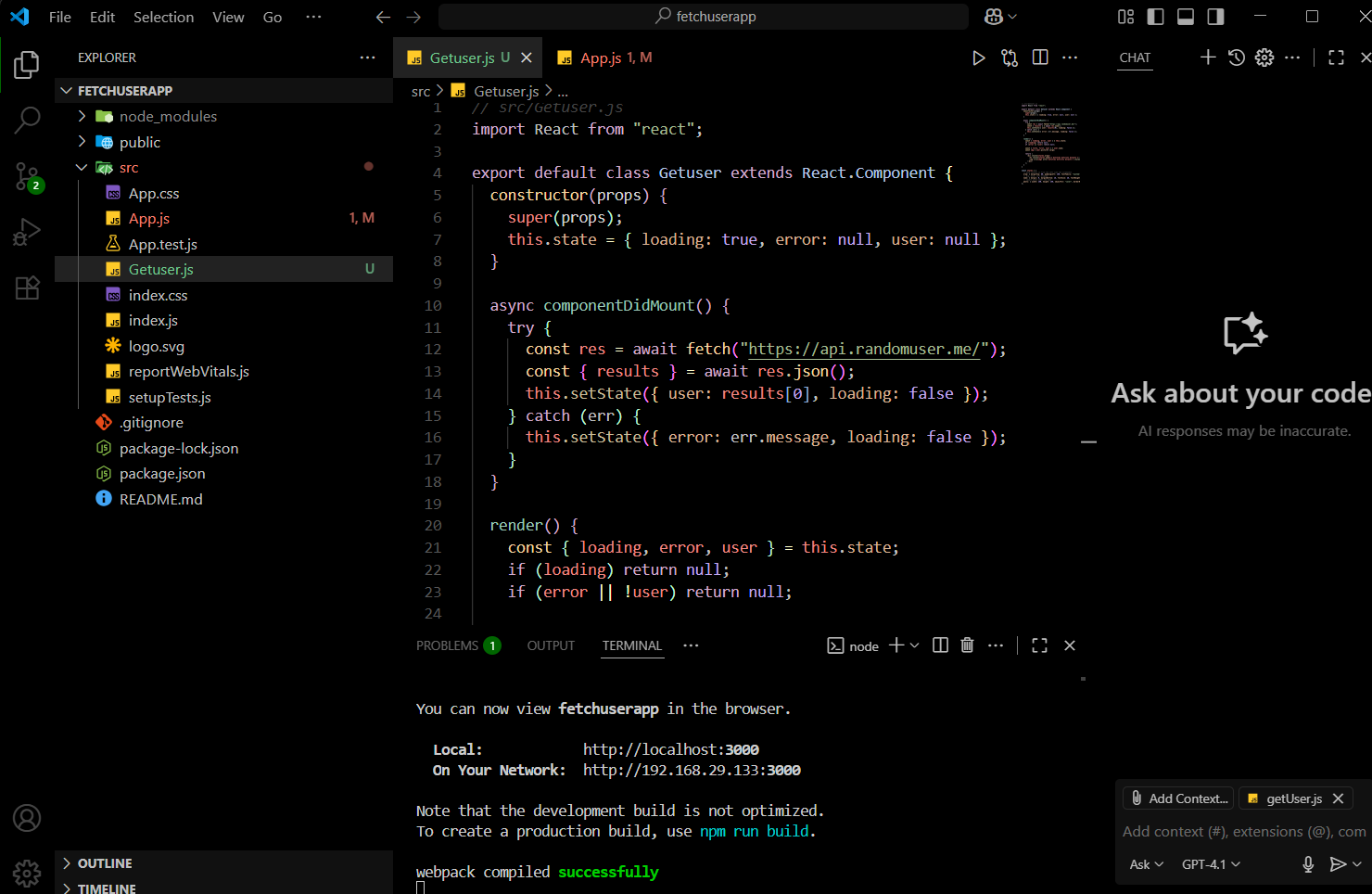
Inline styles are applied only to satisfy the specified visual: the wrapper uses top margin and left padding so the content anchors to the left; the <h1> is bold and large; the image is shown below the heading, sized to a square with fixed width and height, and object-fit: cover to preserve framing. No cards, badges, buttons, or centered content are included, as they are not part of the reference.

After starting the app, the page shows the user’s name as a left-aligned heading with the square photo directly under it. The fields correspond exactly to the API’s title, first, and last name. The visual spacing and alignment match the provided screenshot.

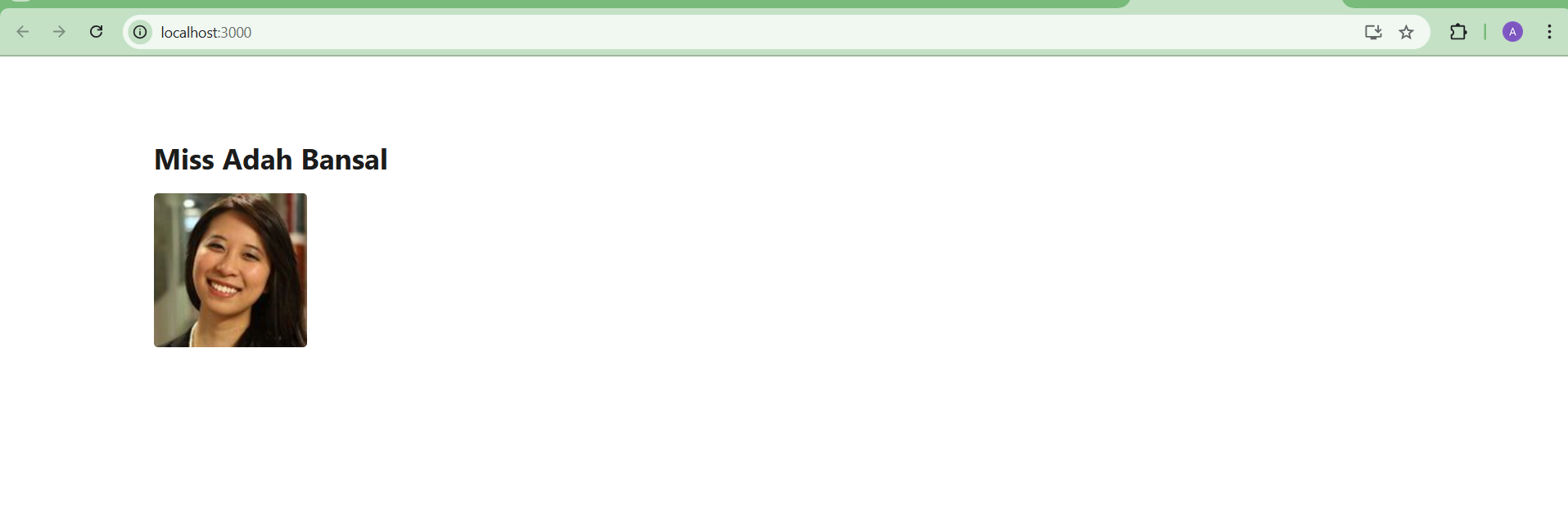
* src/Getuser.js (class component with componentDidMount fetch and render)
* src/App.js (renders <Getuser /> as the application UI)

This completes the handson exactly to specification: class-based lifecycle fetch, single-user render, and the precise left-aligned layout with the name heading and square image beneath it.

****

****

**OUTPUT:**

****