HyperionDev

React – Hooks

Model Answer Approach

Visit our website

Auto-graded task 1

Approach taken:

This approach builds a React application by breaking it into two main components: `NameInput` and `CountryDetails`, with the `App` component managing the overall logic.

- **Component breakdown:**
    - `NameInput` component: Handles user input and triggers the search when the input is submitted.
    - `CountryDetails` component: Displays the country data retrieved from the API based on the user's input.
    - `App` component: Orchestrates the state management and API calls. It uses the `useState` hook to manage state for:
        - `countryData`: stores the data retrieved from the Nationalize.io API.
        - `loading`: keeps track of the loading state during API calls.
        - `error`: handles any potential errors from the API request or network issues.
- **Data fetching:** The `fetchCountryData` function is responsible for making an asynchronous API call to Nationalize.io based on the user's input, using `async/await` syntax for cleaner, more readable code.
- **State management:** The `useState` hook effectively tracks the changes in input, fetched data, errors, and loading state. This makes the application responsive to user actions and ensures proper handling of different scenarios.
- **Error and loading states:** To improve user experience, loading states are displayed while the API is fetching data, and error messages are shown if something goes wrong (e.g., network failure or invalid input).

Why it is a fitting solution:

- **Separation of concerns:**
    - Each component is responsible for a specific task: `NameInput` handles input, `CountryDetails` displays the fetched data, and the `App` component manages application state and logic. This results in a clean and maintainable codebase.

- **User feedback:**
    - By managing loading states and error messages, the application provides a user-friendly and responsive interface. Users are kept informed of the app's status, enhancing the overall experience.
- **Clear and readable code:**
    - The use of `async/await` for API calls keeps the code simple and easy to understand, reducing potential bugs associated with callback-based approaches.

Potential common pitfalls:

- **Error handling:**
    - If API errors or network issues are not handled properly, the application may appear unresponsive. Using a `try/catch` block to catch and display errors ensures the app stays user-friendly even when issues occur.
- **Input validation:**
    - Invalid or empty inputs can cause unwanted behaviour or failed API requests. The input should be trimmed and validated before making the API call to ensure valid results are returned.
- **Loading state management:**
    - Forgetting to display loading indicators while the data is being fetched can lead to confusion for the user. Managing and showing a loading state provides better clarity and enhances the user experience.

Auto-graded task 2

Approach taken:

The code sets up a simple weather app using React and the Weather API. Key functionalities include user input for city names, fetching weather data based on the input, and handling both successful and failed API responses.

- **State management:** The app uses the `useState` hook to manage three states:
    - `city`: to store the user's input for the city name.
    - `forecastData`: to store the weather data received from the Weather API.

    - `error`: to handle error messages when the fetch operation fails or no city is provided.
- **API integration:**
    - The app uses the `fetch` method to call the Weather API and retrieves a 1-day weather forecast for the provided city.
    - The API key is stored securely using environment variables (`import.meta.env.VITE_API_KEY`).
- **Error handling:**
    - If no city name is provided, an error message is displayed prompting the user to enter a city.
    - If the API request fails (due to network issues or invalid data), a descriptive error message is displayed.
- **User interface:**
    - An input field lets users type the city name.
    - A button triggers the `getForecast` function to fetch weather data.
    - If an error occurs or the city name is missing, the app displays the error.
    - When data is available, it shows the forecasted weather conditions (temperature, condition text, and weather icon).

Why it is a fitting solution:

- **State management:** The `useState` hook handles user input, fetched data, and error messages efficiently, making the UI responsive to user actions.
- **Input handling:** By controlling the input with the `value` and `onChange` properties, the app properly manages the user's city input.
- **API data fetching:** The `fetch` function is used to retrieve weather information. Error handling ensures that if a user provides an invalid city or there's a network issue, appropriate messages are shown to the user.
- **Feedback and usability:** Users are given immediate feedback when an error occurs (e.g., invalid city name or API error) and see updated weather information upon successful API responses.

Potential common pitfalls:

- **API errors:** If the city name is misspelt or not recognized by the API, an error will be returned, so it is important to provide user feedback to correct this.
- **Empty input:** If the user attempts to fetch data without entering a city name, the app handles this with an appropriate error message.
- **Exposing API keys:** Using environment variables helps protect the API key, ensuring that the API key is kept secure and is not exposed in the client-side code.