HyperionDev

React – Local State Management and Events

Model Answer Approach

Visit our website

Auto-graded task

1. `Bank` component

Approach taken:

In the `Bank` component, the approach involves creating a functional component that manages the state for various banking operations such as deposits, withdrawals, interest, and fees. The state variables are updated via corresponding functions (`updateDeposit`, `updateWithdrawal`, `addInterest`, and `addFees`). The component layout is structured using Bootstrap for responsive design, with separate child components handling user input and displaying the balance.

Why it's a fitting solution:

This approach encapsulates the core banking functionalities within a single component, providing a clear separation of concerns. It uses state management to handle user inputs and compute the balance dynamically. By leveraging Bootstrap, the component ensures a responsive and user-friendly interface.

Potential common pitfalls:

State management complexity: As the number of inputs and calculations increases, managing state and passing props can become complex. We have to ensure that state updates are handled properly to avoid inconsistent UI.

Calculation accuracy: We need to ensure that calculations for balance, interest, and fees are accurate and consider edge cases such as negative values or invalid inputs.

2. `DisplayBalance` component

Approach taken:

The `DisplayBalance` component is designed to present the current balance to the user. It receives the balance value as a prop and formats it to two decimal places for clarity.

Why it's a fitting solution:

This component focuses solely on displaying the balance, keeping its responsibilities limited and simple. Formatting the balance ensures that it is always presented clearly, enhancing the user experience.

Potential common pitfalls:

Prop validation: We have to ensure that the balance prop is always a number. We can use `PropTypes` to help catch type-related issues but this is not relevant here.

Formatting issues: We can try and handle edge cases where the balance might be zero or negative, and ensure that formatting remains consistent.

3. `Deposit` component

Approach taken:

The `Deposit` component allows users to input deposit amounts and submit them. It uses state to manage the input value and a button to trigger the update. The component ensures that only valid numeric inputs are accepted.

Why it's a fitting solution:

This approach ensures that deposits are handled interactively, with input validation to prevent non-numeric values. Clearing the input field after submission improves user experience.

Potential common pitfalls:

Input validation: We need to ensure that only positive numeric values are accepted, prevent invalid inputs, and check that the input field is cleared after the deposit is processed.

State management: We need to properly handle state updates to avoid inconsistencies in the displayed deposit amount.

4. `Withdrawal` component

Approach taken:

The `Withdrawal` component is similar to the `Deposit` component but focuses on withdrawing amounts. It validates input to ensure it's numeric and non-negative, and triggers an update when the button is clicked.

Why it's a fitting solution:

This component handles user input for withdrawals in a straightforward manner, with validation to ensure only valid values are processed. This approach maintains consistency with other components.

Potential common pitfalls:

Negative values: We need to ensure that users cannot input negative values and that the balance does not become negative.

State clearing: We have to make sure that the input fields are cleared after submission and that state updates are handled correctly.

5. `AddInterest` Component

Approach taken:

The `AddInterest` component allows users to input an interest rate and apply it. It validates the input to ensure it is numeric and non-negative, then updates the parent component with the entered interest value.

Why it’s a fitting solution:

By focusing on input validation and clear submission, this component ensures that interest values are handled correctly. The user interface is kept simple and intuitive.

Potential common pitfalls:

Input validation: We need to ensure that the interest rate is positive and properly formatted, and prevent invalid or extreme values.

State management: We need to confirm that the state is correctly updated and cleared after submission to avoid confusion.