

## Part I: Foundations of React

### Exercise 1: The React Paradigm

#### 1. Conceptual Questions:

- In your own words, describe the difference between an imperative and a declarative approach to UI development. Provide a simple, non-code example (like asking for a coffee).<sup>1</sup>
- List three key benefits of using a component-based architecture. For each benefit, briefly explain why it is advantageous for developing applications.<sup>5</sup>
- Explain the role of the Virtual DOM. How does the "reconciliation" process help improve application performance compared to manipulating the real DOM directly?

11

### Exercise 2: Setting Up a Modern React Development Environment

#### 1. Project Setup:

- Using your terminal, create a new React project named react-basics-exercise using Vite with the command: `npm create vite@latest react-basics-exercise -- --template react`.<sup>17</sup>
- Once the project is created, navigate into the project directory, install the dependencies (`npm install`), and start the development server (`npm run dev`). What URL is your application running on?
- Open the project in your code editor. Identify and describe the purpose of the following files/folders: `index.html`, `src/main.jsx`, and `src/App.jsx`.<sup>20</sup>

---

## Part II: Building with Components

### Exercise 3: Mastering Functional Components and JSX

#### 1. Create a User Profile Component:

- Inside the `src` folder of your Vite project, create a new file named `UserProfile.jsx`.
- In this file, create a functional component named `UserProfile`.

- The component should return the following JSX structure, but wrapped in a single root element using a Fragment (`<>...</>`):<sup>22</sup>

JavaScript

```
<h2>User Profile</h2>
<p>Name: John Doe</p>
<p>Email: john.doe@example.com</p>
```

- Import and render this UserProfile component inside your App.jsx file, replacing the default content.

## 2. Dynamic Data with JSX:

- Inside your UserProfile component, create a JavaScript object to hold user data:

JavaScript

```
const user = {
  name: 'Jane Smith',
  email: 'jane.smith@example.com',
  avatarUrl: 'https://i.imgur.com/yXOvdOSs.jpg',
  imageSize: 90,
};
```

- Modify the JSX to use the data from this object. Use curly braces {} to embed the `user.name` and `user.email`.<sup>23</sup>
- Add an `<img>` tag to display the user's avatar. The `src` attribute should be set to `user.avatarUrl`, and remember to make it a self-closing tag (`<img />`).
- The alt text for the image should be the user's name. The `width` and `height` attributes should both be set to `user.imageSize`.
- Give the `<img>` a CSS class of `profile-avatar`. Remember that the class attribute is written as `className` in JSX.<sup>22</sup>

## Exercise 4: Data Flow with Props

### 1. Passing Props:

- In App.jsx, create two different user objects.
- Render the UserProfile component twice, passing each user object as a prop named `userData`.

JavaScript

```
// In App.jsx
<UserProfile userData={user1} />
<UserProfile userData={user2} />
```

- Modify the UserProfile.jsx component to receive the userData prop. Use prop destructuring in the function signature: function UserProfile({ userData }).<sup>24</sup>
  - Update the component to use the data from userData instead of the hardcoded object.<sup>25</sup>
2. **PropTypes and Default Props:**
- Install the prop-types library in your project: npm install prop-types.
  - In UserProfile.jsx, import PropTypes and add prop validation. The userData prop should be an object with a specific shape: name (a required string) and email (a string).<sup>30</sup>
  - Modify the UserProfile component to also accept a theme prop. Use ES6 default parameters in the function signature to set its default value to 'light'.<sup>34</sup>
  - Add a className to the main wrapper element of the UserProfile component that dynamically changes based on the theme prop (e.g., className={profile-card theme-\${theme}}).
- 

## Part III: State and Interactivity

### Exercise 5: Managing Component Memory with State

1. **Simple Counter:**
- Create a new component file named Counter.jsx.
  - Inside this component, import and use the useState hook to create a state variable called count, initialized to 0.
  - Render the current value of count inside a <p> tag.
  - Add a button that, when clicked, increments the count by 1.<sup>39</sup>
2. **Conceptual Question:**
- Imagine you want to add a button inside the UserProfile component that toggles the user's online status (displaying "Online" or "Offline"). Would you use props or state to manage the online status? Explain your reasoning.<sup>46</sup>

### Exercise 6: Handling User Interaction

1. **Controlled Input:**
- Create a new component Login.jsx.
  - Add a state variable username initialized to an empty string.
  - Create an <input type="text" />.
  - Make it a controlled component by setting its value attribute to the username state variable and its onChange handler to a function that updates the username state with event.target.value.<sup>50</sup>
  - Display the current value of username in a <p> tag below the input to confirm it's

working.

2. **Multi-Input Form:**

- Expand the Login.jsx component to include a password field.
  - Instead of two separate state variables, manage the form data in a single state object: `const [ formData, setFormData ] = useState({ username: "", password: "" })`.
  - Create a single `handleChange` function that can update both fields. Use the `name` attribute on the input elements to dynamically update the correct property in the `formData` object.<sup>51</sup>
  - Add a `<form>` tag with an `onSubmit` handler. The handler should prevent the default form submission and log the `formData` object to the console.<sup>51</sup>
- 

## Part IV: Advanced Composition Patterns

### Exercise 7: Advanced Component Design and Reusability

1. **Wrapper Component with children:**

- Create a Card.jsx component.
- This component should accept a `children` prop and a `title` prop.
- It should render a `div` with a class `card`. Inside, it should render an `<h3>` with the `title`, a horizontal rule `<hr />`, and then the `{children}`.
- In App.jsx, use your new Card component to wrap the UserProfile components you created earlier. Pass a unique title to each card.<sup>24</sup>

2. **Lifting State Up:**

- Create an Accordion.jsx component that will act as the parent.
  - Create a Panel.jsx component that will be the child.
  - **Step 1 (Local State):** Initially, give each Panel its own `isActive` state (a boolean). A panel should show its content if `isActive` is true, and a "Show" button if it's false. Render two panels inside Accordion and verify that they can be opened and closed independently.
  - **Step 2 (Lift State):** Now, modify the components to enforce that only one Panel can be open at a time. To do this, "lift the state up" to the Accordion component.
    - The Accordion component should hold the state for which panel is active (e.g., `const [ activeIndex, setActiveIndex ] = useState(0)` ).
    - The Accordion should pass down an `isActive` prop (a boolean calculated from `activeIndex`, like `isActive={activeIndex === 0}`) and an `onShow` prop (a function to update the `activeIndex`, like `onShow={() => setActiveIndex(0)}`) to each Panel.
    - The Panel component should be modified to be controlled by these props, removing its local state.<sup>25</sup>
-

## Part V: Debugging and Tooling

### Exercise 8: Essential Debugging with React Developer Tools

#### 1. Using the Components Tab:

- Make sure you have the React Developer Tools extension installed in your browser.
- Open the developer tools on your running application and navigate to the "Components" tab.
- Select your Counter component from the component tree.
- In the right-hand panel, find the hooks Exercise. Manually change the value of the count state. Observe the change in the UI.<sup>66</sup>

#### 2. Identifying Re-Renders:

- In the React DevTools settings (gear icon), under the "General" tab, enable "Highlight updates when components render."
  - Interact with your application. For example, click the button in your Counter component or type into your Login form.
  - Observe the colored boxes that appear. Which components are re-rendering when you interact with the Counter? Which components re-render when you type in the Login form?<sup>77</sup>
- 

## Part VI: Capstone Project - Simple To-Do List

This project will integrate all the concepts you've learned: component hierarchy, props, state, event handling, controlled components, and lifting state up. (You can use your own personal project if it fits with requirements.)

### 1. Project Goal & Setup

- **Goal:** Build a functional To-Do List application where you can add, toggle (mark as complete), and delete tasks.
- **Setup:** In your src folder, create the following new component files:
  - TodoApp.jsx (This will be the main parent component for the app)
  - TodoForm.jsx
  - TodoList.jsx
  - TodoItem.jsx
- Render the <TodoApp /> component from your main App.jsx.

### 2. Building the Static Components

First, build the UI with no interactivity.

- **Todoltem.jsx:** Create a component that accepts a todo object (e.g., { id: 1, text: 'Learn React', completed: false }) as a prop and renders its text in an `<li>` element.
- **TodoList.jsx:** Create a component that accepts a todos array as a prop. It should use the `.map()` function to render a Todoltem for each object in the array.
- **TodoForm.jsx:** Create a component that renders a `<form>` with an `<input type="text" />` and a `<button type="submit">Add Todo</button>`.
- **TodoApp.jsx:** This is your main container.
  - Create a hardcoded array of todo objects.
  - Arrange the static layout: render the TodoForm and the TodoList, passing the hardcoded array as a prop to TodoList.

### 3. Adding State and Interactivity

Now, let's make the app dynamic.

- **Controlled Form:** In TodoForm.jsx, use the `useState` hook to manage the value of the input field. Make it a controlled component by linking the input's value and `onChange` attributes to your state.
- **Lifting State Up:** The list of todos is shared data. It needs to live in the TodoApp component.
  - In TodoApp.jsx, move the hardcoded array into a state variable using `useState`.
  - Create a function `addTodo(text)` inside TodoApp that adds a new todo object to the state array. *Hint: You'll need to generate a unique ID for each new todo.*
  - Pass this `addTodo` function as a prop to TodoForm.
  - In TodoForm, modify its `handleSubmit` function to call the `addTodo` prop with the input's current value and then clear the input field.
- **Toggling and Deleting Todos:**
  - In TodoApp.jsx, create two more functions: `toggleTodo(id)` and `deleteTodo(id)`. These functions will update the todos state array by finding the correct todo by its id and either flipping its completed status or removing it from the array.
  - Pass these functions down as props through TodoList to each Todoltem.
  - In Todoltem.jsx, add a checkbox (`<input type="checkbox" />`) and a "Delete" button.
  - Add `onClick` handlers to them that call the `toggleTodo` and `deleteTodo` props, passing the item's own id.
  - Add conditional styling to Todoltem so that the text has a line-through when `todo.completed` is true.

### 4. Debugging with DevTools

- Open the React Developer Tools in your browser.
- Select the TodoApp component in the "Components" tree. Inspect its state and observe the array of todos.
- Use your application to add, toggle, and delete a few todos. Watch how the state updates in real-time in the DevTools panel.

- Enable "Highlight updates when components render." Type in the input field and notice that only the TodoForm re-renders. Add a new todo and observe which components render