React Exercises for BCU students

Author: MSc. Tran Vinh Khiem

Class: BCU2025

Introduction to React

React is a declarative, efficient, and flexible JavaScript library for building user interfaces. It lets you compose complex UIs from small and isolated pieces of code called “components.” Developed by Facebook, React has gained immense popularity for its component-based architecture, virtual DOM, and unidirectional data flow, making it a powerful tool for developing single-page applications (SPAs) and mobile applications.

Why React?

React simplifies the process of building interactive user interfaces. Its core principles allow developers to create reusable UI components, manage application state efficiently, and render updates to the DOM in an optimized manner. This leads to faster development cycles, improved performance, and easier maintenance of complex applications.

Key Concepts Overview

Before diving into practical exercises, it's crucial to understand some fundamental concepts that underpin React's functionality:

- **Components:** The building blocks of any React application. Components are independent, reusable pieces of UI. They can be functional (using JavaScript functions) or class-based (using ES6 classes).
- **JSX (JavaScript XML):** A syntax extension for JavaScript that allows you to write HTML-like code directly within your JavaScript files. JSX makes it easier to describe what the UI should look like.
- **Props (Properties):** A mechanism for passing data from a parent component to a child component. Props are read-only and help in creating dynamic and reusable components.
- **State:** An object that holds data that may change over the lifetime of a component. When the state changes, the component re-renders to reflect the updated data.
- **Virtual DOM:** React maintains a lightweight representation of the actual DOM in memory. When the state of a component changes, React first updates its virtual DOM, then efficiently calculates the differences and updates only the necessary parts of the real DOM, leading to performance improvements.
- **Hooks:** Introduced in React 16.8, Hooks are functions that let you

use state and other React features without writing a class. Key hooks include `useState` for managing state and `useEffect` for handling side effects like data fetching or subscriptions.

This document will guide you through a series of exercises designed to solidify your understanding of these concepts and build practical React development skills.

Section 1: React Fundamentals

This section delves into the foundational concepts of React, which are crucial for building any React application. Understanding these building blocks will enable you to create interactive and dynamic user interfaces.

1.1 Components

At the heart of React are **components**, which are independent and reusable bits of code. They serve the same purpose as JavaScript functions, but work in isolation and return rich HTML. Components come in two main types:

- **Functional Components:** These are JavaScript functions that accept props as an argument and return React elements. They are simpler to write and are the preferred way to write components in modern React due to the introduction of Hooks.

  ```
  // Example of a Functional Component

  function Welcome(props) {

    return <h1>Hello, {props.name}</h1>;

  }
  ```

- **Class Components:** These are ES6 classes that extend `React.Component` and have a `render()` method that returns React elements. While still supported, they are less common in new React development.

  ```
  // Example of a Class Component

  class Welcome extends React.Component {

    render() {

      return <h1>Hello, {this.props.name}</h1>;

    }
  ```

```
}
```

1.2 JSX (JavaScript XML)

JSX is a syntax extension for JavaScript. It is used with React to describe what the UI should look like. JSX might remind you of a template language, but it comes with the full power of JavaScript. It gets compiled into `React.createElement()` calls, which return plain JavaScript objects called “React elements.”

- **Embedding Expressions:** You can embed any JavaScript expression within JSX by enclosing it in curly braces `{}`. This allows for dynamic content rendering.

```
const name = 'World';

const element = <h1>Hello, {name}!</h1>; // Embedding a variable

function formatUser(user) {

  return user.firstName + ' ' + user.lastName;

}

const user = {

  firstName: 'Tran',

  lastName: 'Khiem'

};

const greeting = (

  <h1>

    Hello, {formatUser(user)}!

  </h1>

); // Embedding a function call
```

- **Styling in JSX:** You can apply styles to JSX elements using the `style` attribute, which accepts a JavaScript object with camelCased CSS properties.

```
const myStyle = {
```

```
  color: 'blue',
  fontSize: '16px'
};
const styledElement = <p style={myStyle}>This is a styled paragraph.</p>;
```

1.3 Props (Properties)

Props are arguments passed into React components. Props are passed to components via HTML attributes. They are a way of passing data from parent to child components. Props are read-only, meaning a child component cannot modify the props it receives from its parent. This ensures a unidirectional data flow, making applications easier to understand and debug.

```
// Parent Component
function App() {
  return <Welcome name="Alice" />;
}
// Child Component
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

1.4 State

While props allow data flow from parent to child, **state** is used to manage data that is internal to a component and can change over time. When a component's state changes, React re-renders the component to reflect the updated data. In functional components, state is managed using the `useState` Hook.

```
import React, { useState } from 'react';
function Counter() {
  const [count, setCount] = useState(0); // `count` is the state variable, `setCount` is the function to update it
```

```
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

1.5 Event Handling

React elements handle events using camelCase attribute names, similar to HTML. Event handlers in React are functions that are executed when a specific event occurs (e.g., a button click, input change). You pass a function as the event handler, rather than a string.

```
function MyButton() {
  function handleClick() {
    alert('Button clicked!');
  }
  return (
    <button onClick={handleClick}>
      Click Me
    </button>
  );
}
```

1.6 Conditional Rendering

Conditional rendering in React allows you to render different elements or components based on certain conditions. This is a powerful way to create dynamic UIs that respond to changes in application state or props. You can use JavaScript operators like `if` statements, logical `&&` operator, or the ternary operator (`condition ? true : false`).

```
function UserGreeting(props) {
  return <h1>Welcome back!</h1>;
}
function GuestGreeting(props) {
  return <h1>Please sign up.</h1>;
}
function Greeting(props) {
  const isLoggedIn = props.isLoggedIn;
  if (isLoggedIn) {
    return <UserGreeting />;
  }
  return <GuestGreeting />;
}
// Usage:
// <Greeting isLoggedIn={true} />
// <Greeting isLoggedIn={false} />
```

1.7 List Rendering

When working with collections of data, React provides a straightforward way to render lists of elements. You typically use the JavaScript `map()` array method to iterate over an array and

return a list of React elements. It's crucial to provide a unique `key` prop for each list item to help React efficiently update and re-render lists.

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <li key={number.toString()}>
      {number}
    </li>
  );
  return (
    <ul>{listItems}</ul>
  );
}

const numbers = [1, 2, 3, 4, 5];

// Usage:

// <NumberList numbers={numbers} />
```

Section 2: React Hooks

React Hooks are functions that let you “hook into” React state and lifecycle features from functional components. Hooks do not work inside class components. They were introduced in React 16.8 to provide a more direct API to the React concepts you already know: props, state, context, refs, and lifecycle. Hooks make it easier to reuse stateful logic between components without restructuring your component hierarchy.

2.1 `useState` Hook

The `useState` Hook allows you to add React state to functional components. It returns a pair: the current state value and a function that lets you update it. You can call this function from an

event handler or somewhere else. It's similar to `this.setState` in a class, but it doesn't merge the old and new state together.

```
import React, { useState } from 'react';

function Example() {

  // Declare a new state variable, which we'll call


`count`

  const [count, setCount] = useState(0);

  return (

    <div>

      <p>You clicked {count} times</p>

      <button onClick={() => setCount(count + 1)}>

        Click me

      </button>

    </div>

  );

}
```

2.2 `useEffect` Hook

The `useEffect` Hook lets you perform side effects in functional components. Side effects are operations that affect the outside world, such as data fetching, subscriptions, or manually changing the DOM. `useEffect` runs after every render of the component, but you can control when it runs by providing a dependency array.

```
import React, { useState, useEffect } from 'react';

function DataFetcher() {

  const [data, setData] = useState(null);
```

```
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    // This function runs after every render
    async function fetchData() {
      try {
        const response = await fetch('https://jsonplaceholder.typicode.com/todos/1');
        const json = await response.json();
        setData(json);
      } catch (error) {
        console.error('Error fetching data:', error);
      } finally {
        setLoading(false);
      }
    }
    fetchData();
    // Cleanup function (optional): runs before the component unmounts or before the effect re-runs
    return () => {
      // Any cleanup code here
    };
  }, []); // Empty dependency array means this effect runs only once after the initial render

  if (loading) {
```

```
    return <p>Loading data...</p>;
  }
  return (
    <div>
      <h2>Fetched Data:</h2>
      <p>Title: {data.title}</p>
      <p>Completed: {data.completed ? 'Yes' : 'No'}</p>
    </div>
  );
}
```

2.3 `useContext` Hook

The `useContext` Hook allows you to subscribe to React context without introducing nesting. Context provides a way to pass data through the component tree without having to pass props down manually at every level. This is particularly useful for global data like themes, user authentication, or preferred language.

```
import React, { createContext, useContext, useState } from 'react';

// 1. Create a Context

const ThemeContext = createContext(null);

// 2. Create a Provider component

function ThemeProvider({ children }) {
  const [theme, setTheme] = useState('light');
  const toggleTheme = () => {
    setTheme((prevTheme) => (prevTheme === 'light' ? 'dark' : 'light'));
  };
```

```
  return (
    <ThemeContext.Provider value={{ theme, toggleTheme }}>
      {children}
    </ThemeContext.Provider>
  );
}
// 3. Consume the Context using useContext Hook
function ThemeButton() {
  const { theme, toggleTheme } = useContext(ThemeContext);
  return (
    <button onClick={toggleTheme} style={{
      background: theme === 'light' ? '#fff' : '#333',
      color: theme === 'light' ? '#333' : '#fff',
      padding: '10px',
      borderRadius: '5px',
      cursor: 'pointer'
    }}>
      Toggle Theme ({theme})
    </button>
  );
}
// Example Usage:
```

```
function App() {
  return (
    <ThemeProvider>
      <div style={{
        padding: '20px',
        border: '1px solid #ccc',
        borderRadius: '8px',
        textAlign: 'center'
      }}>
        <h1>Context API Example</h1>
        <ThemeButton />
      </div>
    </ThemeProvider>
  );
}
```

Section 3: React Router and Forms

Building single-page applications (SPAs) often requires managing different views or pages without full page reloads. **React Router** is a popular library that enables declarative routing in React applications. Additionally, handling user input through forms is a common requirement, and React provides mechanisms to manage form state efficiently.

3.1 React Router

React Router is a collection of navigational components that compose declaratively with your application. It allows you to define routes that map to different components, enabling navigation within your SPA. Key components include:

- `<BrowserRouter>`: Uses the HTML5 history API (pushState, replaceState and the popstate event) to keep your UI in sync with the URL.
- `<Routes>`: A new component introduced in React Router v6 that groups individual `<Route>` components.
- `<Route>`: Renders UI when its path matches the current URL.
- `<Link>`: Provides declarative, accessible navigation around your application.

To use React Router, you typically install it via npm:

```
npm install react-router-dom@6
```

```
import React from 'react';

import { BrowserRouter as Router, Routes, Route, Link } from 'react-router-dom';

function Home() {

  return <h2>Home</h2>;

}

function About() {

  return <h2>About</h2>;

}

function Contact() {

  return <h2>Contact</h2>;

}

function App() {

  return (

    <Router>

      <div>

        <nav>
```

```
          <ul>
            <li>
              <Link to="/">Home</Link>
            </li>
            <li>
              <Link to="/about">About</Link>
            </li>
            <li>
              <Link to="/contact">Contact</Link>
            </li>
          </ul>
        </nav>
        {/* A <Routes> looks through its children <Route>s and renders the first one that matches the current URL. */}
        <Routes>
          <Route path="/about" element={<About />} />
          <Route path="/contact" element={<Contact />} />
          <Route path="/" element={<Home />} />
        </Routes>
      </div>
    </Router>
  );
}
```

3.2 Forms

Handling forms in React involves managing the state of form inputs. React uses a concept called

controlled components for form elements. In a controlled component, the form data is handled by the React component. The input element's value is controlled by React state, and any changes to the input are handled by an event handler that updates the state.

Controlled Components

With controlled components, every state mutation will have an associated handler function. This makes it easy to validate or modify the input as the user types.

```
import React, { useState } from 'react';

function NameForm() {

  const [name, setName] = useState('');

  const handleChange = (event) => {

    setName(event.target.value);

  };

  const handleSubmit = (event) => {

    alert('A name was submitted: ' + name);

    event.preventDefault(); // Prevents the default form submission behavior

  };

  return (

    <form onSubmit={handleSubmit}>

      <label>

        Name:

        <input type="text" value={name} onChange={handleChange} />

      </label>
```

```
      <button type="submit">Submit</button>
    </form>
  );
}
```

Handling Multiple Inputs

When you have multiple controlled input elements, you can add a `name` attribute to each element and let the handler function choose what to do based on `event.target.name`.

```
import React, { useState } from 'react';

function ReservationForm() {
  const [state, setState] = useState({
    isGoing: true,
    numberOfGuests: 2,
  });
  const handleInputChange = (event) => {
    const target = event.target;
    const value = target.type === 'checkbox' ? target.checked : target.value;
    const name = target.name;
    setState({
      ...state,
      [name]: value,
    });
  };
  return (
```

```
  <form>
    <label>
      Is going:
      <input
        name="isGoing"
        type="checkbox"
        checked={state.isGoing}
        onChange={handleInputChange}
      />
    </label>
    <br />
    <label>
      Number of guests:
      <input
        name="numberOfGuests"
        type="number"
        value={state.numberOfGuests}
        onChange={handleInputChange}
      />
    </label>
  </form>
);
```

}

Section 4: Context API

Context provides a way to pass data through the component tree without having to pass props down manually at every level. This is particularly useful for global data that many components in an application might need, such as authenticated user, theme (light/dark mode), or preferred language. Context is designed to share data that can be considered “global” for a tree of React components.

When to Use Context

Context is primarily used for sharing data that is considered “global” for a tree of React components, such as the current authenticated user, theme, or preferred language. It avoids the problem of “prop drilling,” where props are passed down through many levels of components that don’t directly use them, just to reach a deeply nested component.

How to Use Context

Using React Context involves three main steps:

1. **Create a Context:** You create a Context object using `React.createContext()`. This object comes with a Provider and a Consumer component.

   const MyContext = React.createContext(defaultValue);

2. **Provide the Context:** The `Provider` component is used to wrap the part of your component tree that needs access to the context. It accepts a `value` prop to be passed to consuming components.

   <MyContext.Provider value={/* some value */}>

   {/* Children components */}

   </MyContext.Provider>

3. **Consume the Context:** Components that need to read the context can use the `useContext` Hook (for functional components) or `MyContext.Consumer` (for class components).

`useContext` Hook (Recommended for Functional Components)

As seen in Section 2.3, the `useContext` Hook is the modern and recommended way to consume context in functional components. It takes a context object (the value returned from `React.createContext`) and returns the current context value for that context.

```
import React, { createContext, useContext, useState } from 'react';

// 1. Create a Context

const ThemeContext = createContext(null);

// 2. Create a Provider component

function ThemeProvider({ children }) {

  const [theme, setTheme] = useState('light');

  const toggleTheme = () => {

    setTheme((prevTheme) => (prevTheme === 'light' ? 'dark' : 'light'));

  };

  return (

    <ThemeContext.Provider value={{ theme, toggleTheme }}>

      {children}

    </ThemeContext.Provider>

  );

}

// 3. Consume the Context using useContext Hook

function ThemeDisplay() {

  const { theme } = useContext(ThemeContext);

  return <p>Current Theme: {theme}</p>;
```

```
}
function ThemeToggle() {
  const { toggleTheme } = useContext(ThemeContext);
  return (
    <button onClick={toggleTheme}>
      Toggle Theme
    </button>
  );
}
// Example Usage:
function App() {
  return (
    <ThemeProvider>
      <div>
        <h1>Context API Example</h1>
        <ThemeDisplay />
        <ThemeToggle />
      </div>
    </ThemeProvider>
  );
}
```

Context is a powerful feature, but it should be used judiciously. For managing component-specific state, `useState` is usually sufficient. Context is best suited for data that is truly global or needed by many components at different nesting levels.

Practical Exercises

This section contains hands-on exercises designed to reinforce the concepts covered in the previous sections. Each exercise includes clear instructions, expected outcomes, and hints to guide you through the implementation. The exercises are arranged in increasing order of complexity, starting with basic component creation and progressing to more advanced topics like routing and context management.

Exercise 1: Hello World Component

Objective: Create your first React functional component that displays a simple greeting message.

Instructions:

1. Create a functional component named `HelloWorld`.
2. The component should return a JSX element that displays "Hello, React World!" inside an `<h1>` tag.
3. Add some basic styling to make the text center-aligned and colored blue.

Expected Output: A webpage displaying "Hello, React World!" as a centered, blue heading.

Hints:

- Use the `style` attribute with a JavaScript object to apply inline styles.
- Remember that CSS properties in JavaScript objects use camelCase (e.g., `textAlign` instead of `text-align`).

Exercise 2: Greeting Card Component

Objective: Create a reusable component that accepts props and displays a personalized greeting.

Instructions:

1. Create a functional component named `GreetingCard`.
2. The component should accept a prop called `name`.
3. Display a greeting message: "Hello, [name]! Welcome to React." inside a styled card.

4. Add CSS styling to create a card-like appearance with padding, border, and shadow.
5. Use the component multiple times with different names to demonstrate reusability.

Expected Output: Multiple greeting cards, each displaying a personalized message for different names.

Hints:

- Access props using the `props` parameter: `props.name`.
- You can destructure props for cleaner code: `function GreetingCard({ name })`.
- Consider using CSS classes or inline styles for the card appearance.

Exercise 3: Counter Application

Objective: Build an interactive counter that demonstrates state management using the `useState` Hook.

Instructions:

1. Create a functional component named `Counter`.
2. Use the `useState` Hook to manage a count state, initialized to 0.
3. Display the current count value.
4. Add three buttons:
    - "Increment" button that increases the count by 1
    - "Decrement" button that decreases the count by 1
    - "Reset" button that sets the count back to 0
5. Style the component to make it visually appealing.

Expected Output: A counter interface with the current count displayed and three functional buttons.

Hints:

- Import `useState` from React: `import React, { useState } from 'react';`.
- Use the state setter function to update the count: `setCount(count + 1)`.
- Event handlers can be inline arrow functions or separate functions.

Exercise 4: Toggle Visibility Component

Objective: Create a component that toggles the visibility of content using conditional rendering.

Instructions:

1. Create a functional component named `ToggleVisibility`.
2. Use `useState` to manage a boolean state for visibility (initially `false`).
3. Add a button labeled "Show Content" or "Hide Content" based on the current visibility state.
4. When visible, display a paragraph with some sample text.
5. When hidden, the paragraph should not be rendered.

Expected Output: A button that toggles between showing and hiding a paragraph of text, with the button text changing accordingly.

Hints:

- Use conditional rendering with the logical `&&` operator or ternary operator.
- The button text should reflect the action it will perform when clicked.

Exercise 5: Basic Todo List

Objective: Build a simple todo list application that demonstrates list rendering and basic state management.

Instructions:

1. Create a functional component named `TodoList`.
2. Use `useState` to manage an array of todo items (start with a few sample items).
3. Display the list of todos using the `map()` function.
4. Add an input field and a button to add new todos to the list.
5. Each todo item should display the text and have a unique key.

Expected Output: A todo list displaying existing items with the ability to add new items through an input field.

Hints:

- Use a separate state for the input field value.
- Remember to provide a unique `key` prop for each list item.
- Clear the input field after adding a new todo.

Exercise 6: Dynamic List Filtering

Objective: Enhance the todo list with filtering capabilities to demonstrate more advanced state management.

Instructions:

1. Extend the previous TodoList component or create a new `FilterableTodoList` component.
2. Add a search input field above the todo list.
3. Filter the displayed todos based on the search input (case-insensitive).
4. Show only todos that contain the search term.
5. Display a message when no todos match the search criteria.

Expected Output: A todo list with a search box that filters the displayed items in real-time as the user types.

Hints:

- Use the `filter()` method to create a filtered array of todos.
- Convert both the search term and todo text to lowercase for case-insensitive matching.
- Use the `includes()` method to check if a todo contains the search term.

Exercise 7: Timer/Stopwatch Component

Objective: Create a timer component that demonstrates the `useEffect` Hook for handling side effects.

Instructions:

1. Create a functional component named `Timer`.
2. Use `useState` to manage the elapsed time in seconds (start at 0).
3. Use `useEffect` to set up an interval that increments the time every second.
4. Add buttons to start, pause, and reset the timer.
5. Display the time in a readable format (e.g., "00:05" for 5 seconds).
6. Ensure the interval is cleaned up when the component unmounts.

Expected Output: A functional stopwatch with start, pause, and reset capabilities, displaying time in MM:SS format.

Hints:

- Use `setInterval` inside `useEffect` to update the time.
- Return a cleanup function from `useEffect` to clear the interval.
- Use the dependency array to control when the effect runs.

Exercise 8: Data Fetching Component

Objective: Build a component that fetches data from an external API using `useEffect`.

Instructions:

1. Create a functional component named `UserProfile`.
2. Use `useState` to manage user data and loading state.
3. Use `useEffect` to fetch user data from JSONPlaceholder API (`https://jsonplaceholder.typicode.com/users/1`).
4. Display a loading message while data is being fetched.
5. Once loaded, display the user's name, email, and website.
6. Handle potential errors gracefully.

Expected Output: A component that shows a loading state, then displays user information fetched from the API.

Hints:

- Use the `fetch()` API or axios for making HTTP requests.
- Handle the asynchronous nature of API calls with async/await or promises.
- Consider using try-catch blocks for error handling.

Exercise 9: Simple Navigation with React Router

Objective: Create a multi-page application using React Router for navigation.

Instructions:

1. Install React Router: `npm install react-router-dom@6`.
2. Create three components: `Home`, `About`, and `Contact`.
3. Set up routing using `BrowserRouter`, `Routes`, and `Route`.
4. Create a navigation menu with links to each page.
5. Each page should display unique content.
6. Style the navigation to highlight the current page.

Expected Output: A single-page application with navigation between three different views without page reloads.

Hints:

- Wrap your app with `BrowserRouter`.
- Use `Link` components for navigation instead of anchor tags.
- Consider using `NavLink` for active link styling.

Exercise 10: Login Form with Validation

Objective: Build a controlled form component with input validation.

Instructions:

1. Create a functional component named `LoginForm`.
2. Use controlled components for email and password inputs.
3. Implement basic validation:
    - Email should contain "@" symbol
    - Password should be at least 6 characters long
4. Display validation errors below each input field.
5. Disable the submit button until all validations pass.
6. Show a success message upon successful form submission.

Expected Output: A login form with real-time validation feedback and conditional submit button enabling.

Hints:

- Use separate state variables for each input and validation errors.
- Validate inputs on every change or on blur events.
- Use conditional rendering to show/hide error messages.

Exercise 11: Theme Switcher with Context API

Objective: Implement a global theme switcher using React Context API.

Instructions:

1. Create a `ThemeContext` using `createContext()`.
2. Build a `ThemeProvider` component that manages theme state (light/dark).
3. Create a `ThemeToggle` component that allows switching between themes.
4. Apply theme-based styling to multiple components.
5. Ensure the theme persists across different components without prop drilling.

Expected Output: An application where users can toggle between light and dark themes, with the change reflected across all components.

Hints:

- Define theme objects with colors for light and dark modes.
- Use the `useContext` Hook to consume the theme in child components.
- Consider storing theme preference in localStorage for persistence.

Homework 1: Simple E-commerce Product Listing (Capstone Project)

Objective: Build a comprehensive application that integrates multiple React concepts learned throughout the course.

Instructions:

1. Create a product listing page that displays a grid of products.
2. Each product should show an image, name, price, and "Add to Cart" button.
3. Implement a shopping cart that tracks added items and quantities.
4. Add filtering capabilities (by category or price range).
5. Use React Router to create separate pages for product details.
6. Use Context API for global cart state management.
7. Include a checkout page that displays cart contents and total price.

Expected Output: A functional e-commerce application with product browsing, cart management, and basic checkout functionality.

Hints:

- Use mock data for products or fetch from a free API like FakeStore API.
- Break down the application into smaller, reusable components.
- Consider using localStorage to persist cart data.
- Implement proper error handling and loading states.

This capstone project integrates components, props, state, hooks, routing, forms, and context management, providing a comprehensive practical application of React concepts.

Solutions and Implementation Guide

This section provides complete solutions for all exercises, along with detailed explanations of the implementation approach. Each solution demonstrates best practices and includes comments to help you understand the code structure and React concepts being applied.

Getting Started

Before working on these exercises, ensure you have the following setup:

Prerequisites

- Node.js (version 14 or higher)
- npm or yarn package manager
- Basic knowledge of JavaScript ES6+ features
- A code editor (VS Code recommended)

Setting Up a React Project

To create a new React application for these exercises:

```
npx create-react-app react-exercises

cd react-exercises

npm start
```

For exercises that require React Router, install it separately:

```
npm install react-router-dom@6
```

Exercise Solutions Overview

The solutions provided demonstrate progressive complexity, starting with basic component creation and advancing to comprehensive applications that integrate multiple React concepts. Each solution includes:

- Complete, functional code
- Inline comments explaining key concepts
- Styling for a professional appearance
- Error handling where appropriate
- Best practices for React development

Key Learning Outcomes

By completing these exercises and studying the solutions, students will gain practical experience with:

Fundamental React Concepts:

- Component creation and composition
- JSX syntax and JavaScript integration
- Props for data passing between components
- State management with the useState Hook
- Event handling and user interactions

Advanced React Features:

- useEffect Hook for side effects and lifecycle management
- Context API for global state management
- React Router for single-page application navigation
- Form handling with controlled components
- API integration and data fetching

Development Best Practices:

- Component organization and reusability
- State management patterns
- Error handling and user feedback
- Performance considerations
- Code structure and maintainability

Implementation Notes

Each solution is designed to be self-contained and can be implemented independently. However, the exercises build upon each other conceptually, so it's recommended to work through them in order. The solutions demonstrate real-world patterns that students will encounter in professional React development.

The styling in these solutions uses inline styles for simplicity and to keep each component self-contained. In production applications, you would typically use CSS modules, styled-components, or CSS frameworks like Tailwind CSS for more maintainable styling solutions.

Testing Your Solutions

When implementing these exercises, consider the following testing approaches:

- **Manual Testing:** Interact with your components to ensure they behave as expected
- **Console Logging:** Use console.log() to debug state changes and data flow
- **React Developer Tools:** Install the React DevTools browser extension to inspect component state and props
- **Error Boundaries:** Implement error boundaries for production-ready applications

Next Steps

After completing these exercises, students should be well-prepared to:

- Build more complex React applications
- Integrate with backend APIs
- Implement state management libraries like Redux
- Explore React frameworks like Next.js
- Contribute to open-source React projects

The capstone e-commerce project (Exercise 12) serves as a comprehensive example that integrates all learned concepts and provides a foundation for building real-world applications.
