ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION)

MODULE 2 (WEEK 3-4)
SETUP RUNTIME ENVIRONMENT AND INSTALL NECESSARY SOFTWARE TO RUN REACT.JS ON YOUR COMPUTER

CHAPTER 1: INTRODUCTION TO REACT.JS DEVELOPMENT ENVIRONMENT

1.1 What is React.js?

React.js is a popular **JavaScript library** used for building **user interfaces (UIs)**, particularly **single-page applications (SPAs)**. It allows developers to create **reusable components**, improving code efficiency and maintainability.

To start developing in React.js, you need to set up a **runtime environment** with the necessary software tools.

1.2 Why Set Up a Proper Development Environment?

- ✅ Enables **faster and efficient coding**.
- ✅ Provides **real-time debugging and error handling**.
- ✅ Ensures compatibility with the latest **React features**.
- ✅ Helps in **managing project dependencies**.

CHAPTER 2: INSTALLING NECESSARY SOFTWARE FOR REACT.JS DEVELOPMENT

2.1 Install Node.js and npm (Node Package Manager)

React.js requires **Node.js**, which provides a runtime for executing JavaScript outside the browser. It also includes **npm**, which helps install React and other dependencies.

Step 1: Download and Install Node.js

1. Go to the official website: https://nodejs.org/
2. Download the **LTS (Long-Term Support) version** (recommended for stability).
3. Run the installer and follow the on-screen instructions.

Step 2: Verify Installation

Once installed, open the terminal or command prompt and check the versions:

```
node -v   # Checks Node.js version

npm -v    # Checks npm version
```

✅ **If both commands return version numbers, Node.js and npm are successfully installed.**

2.2 Install a Code Editor (VS Code Recommended)

A **code editor** helps in writing, debugging, and managing React.js projects efficiently.

Step 1: Download and Install VS Code

1. Visit https://code.visualstudio.com/
2. Click **Download for Windows/Mac/Linux**.
3. Run the installer and follow the setup instructions.

Step 2: Install Essential Extensions for React Development

Open **VS Code**, go to **Extensions (Ctrl + Shift + X)**, and install:
✅ **ES7+ React/Redux/React-Native Snippets** (Shortcuts for React.js boilerplate code).
✅ **Prettier - Code Formatter** (Automatically formats React.js code).
✅ **React Developer Tools** (Helps debug React applications).

2.3 Install Git for Version Control

Git helps in managing code versions and collaborating with teams.

Step 1: Download and Install Git

1. Visit https://git-scm.com/downloads
2. Download and install Git based on your OS.

Step 2: Verify Installation

Check if Git is installed using:

```
git --version
```

✅ **If Git returns a version number, it is installed correctly.**

Chapter 3: CREATING A REACT.JS APPLICATION

3.1 Install Create-React-App (CRA) for Quick Setup

React provides a command-line tool called **Create-React-App (CRA)** to set up projects quickly.

Step 1: Install Create-React-App Globally

npm install -g create-react-app

Step 2: Create a New React App

npx create-react-app my-react-app

✅ This command will **automatically configure** a React project with the required dependencies.

Step 3: Navigate to the Project Folder

cd my-react-app

Step 4: Start the Development Server

npm start

✅ This launches the app in your **default web browser** at http://localhost:3000/.

3.2 Understanding the Project Structure

When you create a React app, the folder structure looks like this:

my-react-app/

```
| -- node_modules/      # Installed dependencies
| -- public/           # Public assets (index.html, favicon)
| -- src/              # Source code (components, styles)
```

```
|-- .gitignore        # Files ignored by Git
|-- package.json      # Project configuration
|-- README.md         # Project documentation
```

Chapter 4: Installing Additional Libraries for React Development

4.1 Install React Router for Navigation

React Router is used to create multi-page navigation within an app.

npm install react-router-dom

4.2 Install Styled-components for Styling

Styled-components allow writing **CSS inside JavaScript** for scoped styling.

npm install styled-components

4.3 Install Axios for API Requests

Axios is a promise-based library for making **HTTP requests** in React.

npm install axios

Chapter 5: Debugging React Applications

5.1 Use React Developer Tools

1. Install **React Developer Tools** in Chrome/Firefox from the Extension Store.
2. Open DevTools (F12 or Ctrl + Shift + I).

3. Navigate to the **React tab** to inspect components and state.

5.2 Debugging in VS Code

✅ Use **console.log()** to check values inside React components.
✅ Open **VS Code's built-in debugger** to set breakpoints.
✅ Use **Error Boundaries** in React to catch and handle errors.

CHAPTER 6: HANDS-ON PRACTICE & EXERCISES

Exercise 1: Create a Simple React App

1. Install Node.js, npm, and VS Code.
2. Set up a new React project using create-react-app.
3. Start the development server and check http://localhost:3000/.

Exercise 2: Modify the Default React App

1. Go to src/App.js.
2. Replace the default content with:

```
function App() {
  return <h1>Hello, React World!</h1>;
}

export default App;
```

Save and see the changes in the browser.

Exercise 3: Install and Use React Router

1. Install React Router:
2. npm install react-router-dom
3. Modify src/App.js to add navigation:

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


function Home() {
  return <h1>Home Page</h1>;
}


function About() {
  return <h1>About Page</h1>;
}


function App() {
  return (
    <Router>
      <nav>
        <Link to="/">Home</Link> | <Link to="/about">About</Link>
      </nav>
      <Routes>
```

```
            <Route path="/" element={<Home />} />
            <Route path="/about" element={<About />} />
          </Routes>
        </Router>
      );
    }

    export default App;
```

Run npm start and test navigation between pages.

CONCLUSION

Key Takeaways

✅ **Node.js and npm** are required to run React.js.
✅ **VS Code is the recommended code editor** for React development.
✅ **Git enables version control and collaboration**.
✅ **Create-React-App (CRA) provides a quick setup**.
✅ **React Router, Styled-components, and Axios** improve app functionality.

🚀 **Next Steps:**

- Learn **React state management (useState, Redux)**.
- Explore **React Hooks and API integration**.

- Build a **real-world React project** (e.g., To-Do App, Weather App).

By completing this **hands-on setup**, you are now ready to **build and deploy React.js applications**! 🎉🚀

REACT FUNDAMENTALS – INTRODUCTION TO REACT & JSX

CHAPTER 1: INTRODUCTION TO REACT

1.1 What is React?

React is a **JavaScript library** used for building **user interfaces (UIs)**, particularly **single-page applications (SPAs)**. Developed by **Facebook (now Meta)**, React simplifies the process of creating **interactive and reusable UI components**.

React is widely used by major companies such as **Netflix, Airbnb, Instagram, and Uber** due to its **speed, scalability, and component-based architecture**.

1.2 WHY USE REACT?

✅ **Component-Based Architecture** – Encourages reusable UI components, making development faster.
✅ **Virtual DOM** – Optimizes rendering, making React applications fast.
✅ **One-Way Data Binding** – Enhances maintainability and predictability.
✅ **Declarative UI** – React updates the UI efficiently when state changes.
✅ **Large Ecosystem** – Includes React Router, Redux, and Material-UI for advanced development.
✅ **SEO Friendly** – Works well with **Server-Side Rendering (SSR)** using Next.js.

1.3 Industry Use Cases of React

🚀 **Social Media Platforms** – Facebook and Instagram use React for dynamic feeds.
🛒 **E-commerce Websites** – Amazon and Shopify integrate React for faster product pages.
📺 **Streaming Services** – Netflix and Disney+ optimize performance using React.
📊 **Dashboards & Admin Panels** – Business analytics tools rely on React for interactive UI components.

CHAPTER 2: GETTING STARTED WITH REACT

2.1 Installing React and Setting Up a Project

React can be **added to an existing project** or **set up from scratch** using Create React App.

Option 1: Setting Up React with Create React App (Recommended)

1. **Install Node.js** (Download from https://nodejs.org/)
2. **Check Node.js and npm version** in the terminal:
3. node -v
4. npm -v
5. **Create a React App** using the command:
6. npx create-react-app my-app
7. **Navigate into the project folder**:
8. cd my-app
9. **Start the React Development Server**:

10. npm start
    - Your app will run at http://localhost:3000/.

2.2 React Folder Structure Overview

/my-app

```
├── /node_modules   (Project dependencies)
├── /public         (Static assets like index.html)
├── /src            (Main React files)
├── index.js        (Entry point of the app)
├── App.js          (Main component file)
├── package.json    (Project configuration & dependencies)
└── README.md       (Documentation)
```

2.3 Understanding Components in React

React applications are built using **components**.

- **Functional Components**: The modern way to write components using functions.
- **Class Components**: Older method using ES6 classes.

Example of a Functional Component (App.js)

```
import React from "react";
```

```
function Welcome() {
  return <h1>Hello, Welcome to React!</h1>;
}

export default Welcome;
```

Example of a Class Component (Less Common in Modern React)

```
import React, { Component } from "react";

class Welcome extends Component {
  render() {
    return <h1>Hello, Welcome to React!</h1>;
  }
}

export default Welcome;
```

CHAPTER 3: INTRODUCTION TO JSX (JAVASCRIPT XML)

3.1 What is JSX?

JSX (JavaScript XML) is a **syntax extension for JavaScript** that allows developers to **write HTML-like code inside JavaScript files**. It makes React components **more readable and expressive**.

Example Without JSX

Using plain JavaScript to create an element:

```
const element = React.createElement("h1", {}, "Hello, World!");
```

Example With JSX

Using JSX to achieve the same result:

```
const element = <h1>Hello, World!</h1>;
```

JSX is **not a template language**; it is **compiled into JavaScript** by Babel.

3.2 Why Use JSX?

✅ **Simplifies UI Code** – Makes React components more readable.
✅ **Better Debugging** – Errors are easier to trace in JSX.
✅ **Performance Optimization** – JSX compiles into optimized JavaScript code.
✅ **JavaScript Integration** – You can use variables, functions, and expressions inside JSX.

3.3 Embedding JavaScript in JSX

JSX allows the use of JavaScript expressions inside curly braces {}.

Example of Using Variables in JSX

```
const name = "John Doe";

const element = <h1>Welcome, {name}!</h1>;
```

Example of Using JavaScript Functions in JSX

```
function getGreeting(user) {
```

```
  return user ? <h1>Hello, {user}!</h1> : <h1>Hello, Guest!</h1>;
}
```

Example of JSX with Conditional Rendering

```
const isLoggedIn = true;
const message = isLoggedIn ? <h1>Welcome Back!</h1> :
<h1>Please Sign In</h1>;
```

3.4 Adding Styles in JSX

CSS can be applied **directly in JSX** using the style attribute (which accepts an object).

Example of Inline Styling in JSX

```
const headingStyle = { color: "blue", fontSize: "24px" };

const element = <h1 style={headingStyle}>Styled Text</h1>;
```

Example of External CSS in React

```
import "./App.css";

function App() {
  return <h1 className="heading">Hello, React!</h1>;
}

/* App.css */
.heading {
```

```
  color: red;
  font-size: 24px;
}
```

CHAPTER 4: HANDS-ON PRACTICE & ASSIGNMENTS

4.1 Hands-on Practice

✅ **Create a React App** using create-react-app.
✅ **Modify App.js to display "Hello, World!"**.
✅ **Use JSX to display a welcome message with your name**.
✅ **Create a component called Profile that displays a user's name and bio**.

4.2 Assignment: Build a React Greeting Component

Task:

1. Create a **functional React component** called Greeting.js.
2. Display a **welcome message** dynamically based on user input.
3. Use **inline CSS styling** to change text color.

Expected Output:

If userName = "Alice", the component should render:

Hello, Alice! Welcome to React.

Solution Example (Greeting.js)

```
import React from "react";
```

```
function Greeting() {
  const userName = "Alice";
  return <h1 style={{ color: "blue" }}>Hello, {userName}! Welcome to React.</h1>;
}

export default Greeting;
```

Chapter 5: Case Study – How React Helped Netflix Optimize Performance

Problem:

Netflix needed a **faster and more scalable UI** for their streaming platform.

Solution:

They **migrated to React** and implemented:
✅ **Component-based architecture** – Reusable UI elements reduced development time.
✅ **Virtual DOM** – Improved performance and reduced re-rendering.
✅ **Lazy loading** – Faster initial page load times.

Result:

- Netflix **reduced load times by 40%**.
- UI became **more responsive and user-friendly**.

CONCLUSION

🚀 React is a powerful **library for building modern web applications**. By mastering **JSX and components,** developers can create **scalable and high-performance applications**.

Next Steps:

✅ Explore **React Hooks (useState, useEffect)**.
✅ Learn **React Router for navigation**.
✅ Build a **complete portfolio website using React**.

By applying these **React fundamentals,** you are on your way to becoming a **React developer!** 🎉

REACT.JS – FUNCTIONAL COMPONENTS & PROPS

CHAPTER 1: INTRODUCTION TO REACT.JS

1.1 What is React.js?

React.js is a **JavaScript library** for building **interactive user interfaces (UIs)**. It follows a **component-based architecture,** making it easy to develop, manage, and scale web applications.

✅ **Developed by Facebook** – Used in large-scale applications like Instagram and WhatsApp Web.
✅ **Component-Based Architecture** – Build reusable UI components.
✅ **Virtual DOM (Document Object Model)** – Improves performance by updating only the changed parts of the UI.
✅ **Fast & Scalable** – Used in modern **single-page applications (SPAs)**.

1.2 Industry Use Cases of React.js

🚀 **E-commerce Websites** – Dynamic product listings and shopping carts.
🚀 **Social Media Platforms** – Facebook, Instagram, and Twitter use React for UI rendering.
🚀 **Dashboard & Analytics Tools** – Interactive charts and real-time data visualization.
🚀 **Job Portals & Online Marketplaces** – LinkedIn, Upwork, and Freelancer use React for seamless user experiences.

CHAPTER 2: FUNCTIONAL COMPONENTS IN REACT.JS

2.1 What are Functional Components?

Functional components are **JavaScript functions** that return **JSX (JavaScript XML)**, used to render UI elements. Unlike class components, functional components:

✅ **Are simple functions** that take input (props) and return JSX.
✅ **Have no this keyword**, making them easier to read.
✅ **Support React Hooks** like useState and useEffect for managing state and side effects.

2.2 Syntax of a Functional Component

A functional component **returns JSX**, which looks like HTML but allows embedding JavaScript:

```
import React from "react";


function Greeting() {
 return <h1>Hello, Welcome to React.js!</h1>;
}


export default Greeting;
```

2.3 Rendering a Functional Component in React

To display this component, import and use it inside App.js:

```
import React from "react";

import Greeting from "./Greeting";
```

```
function App() {
 return (
  <div>
   <Greeting />
  </div>
 );
}

export default App;
```

2.4 Example: Creating a Simple Functional Component

```
import React from "react";

function Welcome() {
 return (
  <div>
   <h1>Welcome to My Website</h1>
   <p>This website is built using React.js.</p>
  </div>
 );
}
```

```
export default Welcome;
```

CHAPTER 3: UNDERSTANDING PROPS IN REACT.JS

3.1 What are Props?

Props (short for "Properties") allow passing **data** from one component to another in React. They are **read-only** and cannot be modified inside the child component.

- ✅ **Used to make components dynamic**.
- ✅ **Passed from parent to child** as function arguments.
- ✅ **Immutable** – They cannot be changed inside the component.

3.2 Passing Props to Functional Components

Props are passed as **attributes** when calling a component:

```
import React from "react";


function Greeting(props) {
  return <h1>Hello, {props.name}!</h1>;
}


export default Greeting;
```

To use this component inside App.js:

```
import React from "react";
```

```
import Greeting from "./Greeting";


function App() {
  return (
    <div>
      <Greeting name="Alice" />
      <Greeting name="Bob" />
    </div>
  );
}


export default App;
```

3.3 Destructuring Props in Functional Components

Instead of using props.name, you can **destructure** props directly:

```
function Greeting({ name }) {
  return <h1>Hello, {name}!</h1>;
}
```

3.4 Example: Creating a Dynamic Card Component using Props

This example demonstrates how to pass multiple props:

```
import React from "react";


function UserCard({ name, age, city }) {
  return (
    <div style={{ border: "1px solid #ccc", padding: "10px", margin: "10px" }}>
      <h2>{name}</h2>
      <p>Age: {age}</p>
      <p>City: {city}</p>
    </div>
  );
}


export default UserCard;
```

To render multiple users:

```
import React from "react";
import UserCard from "./UserCard";


function App() {
  return (
    <div>
```

```
    <UserCard name="Alice" age="25" city="New York" />
    <UserCard name="Bob" age="30" city="Los Angeles" />
    <UserCard name="Charlie" age="28" city="Chicago" />
   </div>
  );
}

export default App;
```

CHAPTER 4: PROPS VS. STATE

4.1 Key Differences Between Props and State

Feature	Props	State
Mutability	Immutable (cannot be changed inside the component)	Mutable (can be changed using useState)
Passed By	Parent component	Managed within the component
Functionality	Used for **data transfer** between components	Used for **component-level data storage**
Example Use Case	User details in a profile card	Toggling dark/light mode

4.2 Using Props and State Together

Example: A counter where a **parent component passes a title (prop),** and the **child component manages count (state)**.

Parent Component (App.js)

```
import React from "react";

import Counter from "./Counter";


function App() {

 return <Counter title="Click Counter" />;

}


export default App;
```

Child Component (Counter.js)

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


function Counter({ title }) {

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


 return (

  <div>

   <h2>{title}</h2>

   <p>Count: {count}</p>
```

```
    <button onClick={() => setCount(count + 1)}>Increment</button>
  </div>
 );
}

export default Counter;
```

CHAPTER 5: HANDS-ON EXERCISES

5.1 Practice Exercises

1. **Create a Functional Component named Message** that takes a text prop and displays it inside a <p> tag.
2. **Modify the UserCard component** to accept an image prop and display the user's profile picture.
3. **Create a Product component** that receives name, price, and description as props and displays them in a card layout.

5.2 Case Study: Props in E-commerce Websites

An **online store** uses **React components with props** to display **dynamic product information**. Instead of hardcoding product details, each product is **fetched from a database and passed as props** to a ProductCard component. This allows thousands of products to be displayed dynamically, improving performance and code efficiency.

CONCLUSION

🎯 Functional components and props are **core concepts in React.js** that enable building **reusable and dynamic UIs**. By understanding these concepts, developers can efficiently manage UI rendering and **create scalable applications**.

🚀 **Next Steps:**

✓ Implement **more complex components** with **props and state**.

✓ Explore **React Hooks** (useState, useEffect).

✓ Work on a **React project** like a **to-do list or a portfolio site**.

By mastering **functional components and props**, you're on your way to becoming a **proficient React developer**! 🔥

HANDLING EVENTS & FORMS IN REACT

CHAPTER 1: INTRODUCTION TO HANDLING EVENTS IN REACT

1.1 What are Events in React?

Events in React are **user interactions** like **clicks, keypresses, form submissions, and mouse movements** that trigger functions. React **event handling** is similar to handling events in JavaScript but follows a **synthetic event system**, ensuring **cross-browser compatibility**.

1.2 Why Use Synthetic Events in React?

React uses **SyntheticEvent**, a wrapper around native events, for:

✅ **Improved Performance** – React reuses event objects.

✅ **Cross-Browser Compatibility** – React normalizes event behavior.

✅ **Prevention of Memory Leaks** – Automatic event pooling saves resources.

1.3 Basic Event Handling in React

In React, event handlers are added as **JSX attributes** inside elements. Instead of onclick, onchange, etc., React uses **camelCase event names** like onClick, onChange, etc.

Example: Handling a Click Event

```
import React from "react";


function ClickHandler() {
  const handleClick = () => {
    alert("Button Clicked!");
```

```
  };

  return <button onClick={handleClick}>Click Me</button>;
}

export default ClickHandler;
```

💡 **Note:** Unlike vanilla JavaScript, React **does not require addEventListener()**. Instead, event handlers are passed as functions inside JSX.

CHAPTER 2: PASSING ARGUMENTS TO EVENT HANDLERS

2.1 Passing Parameters to Functions

To pass parameters inside an event handler, use an **arrow function** or .bind().

Example 1: Passing Arguments Using Arrow Function

```
function GreetUser() {
  const greet = (name) => {
    alert(`Hello, ${name}!`);
  };

  return <button onClick={() => greet("Alice")}>Greet</button>;
}
```

Example 2: Using .bind() Method

```
function ShowMessage() {
   const displayMessage = (message) => {
      alert(message);
   };

   return <button onClick={displayMessage.bind(this, "Welcome to React!")}>Show Message</button>;
}
```

CHAPTER 3: HANDLING FORMS IN REACT

3.1 Why Use Forms in React?

Forms are essential for **user input,** used in login pages, search bars, and checkout processes. React forms require **controlled components** to **manage form state** efficiently.

3.2 Controlled vs. Uncontrolled Components

Controlled Component	**Uncontrolled Component**
Stores form data in **state**	Uses **DOM ref** to access values
Updates values via **onChange**	Uses **defaultValue** for initial values
Ideal for **React apps**	Useful for **simple forms**

3.3 Creating a Controlled Form

React uses **state** to control form inputs dynamically.

Example: Handling Input Fields with useState

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


function SimpleForm() {
  const [name, setName] = useState("");


  const handleChange = (event) => {
    setName(event.target.value);
  };


  const handleSubmit = (event) => {
    event.preventDefault();
    alert(`Submitted Name: ${name}`);
  };


  return (
    <form onSubmit={handleSubmit}>
      <label>
        Enter Name:
        <input type="text" value={name} onChange={handleChange}
/>
```

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

export default SimpleForm;
```

💡 **Key Takeaways:**
✅ **State is updated on every keystroke.**
✅ **onChange updates input value dynamically.**
✅ **handleSubmit prevents default form behavior.**

CHAPTER 4: HANDLING MULTIPLE INPUTS IN FORMS

4.1 Managing Multiple Input Fields

Instead of creating separate states for each input, use a **single object** to store multiple form values.

Example: Handling Multiple Fields with useState

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

function MultiInputForm() {
  const [formData, setFormData] = useState({
    username: "",
```

```
    email: "",
  });

  const handleChange = (event) => {
    setFormData({
      ...formData,
      [event.target.name]: event.target.value,
    });
  };

  const handleSubmit = (event) => {
    event.preventDefault();
    alert(`Username: ${formData.username}, Email:
${formData.email}`);
  };

  return (
    <form onSubmit={handleSubmit}>
      <label>
        Username:
        <input type="text" name="username"
value={formData.username} onChange={handleChange} />
```

```
        </label>
        <br />
        <label>
          Email:
          <input type="email" name="email" value={formData.email}
onChange={handleChange} />
        </label>
        <br />
        <button type="submit">Submit</button>
      </form>
    );
}

export default MultiInputForm;
```

💡 **Best Practices:**
✅ Use **one state object** for multiple inputs.
✅ Spread ...formData to retain existing values while updating fields.

CHAPTER 5: HANDLING CHECKBOXES, RADIO BUTTONS, AND DROPDOWNS

5.1 Handling Checkboxes

Checkbox values are stored as **true/false** (boolean).

Example: Handling Checkboxes in React

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


function CheckboxForm() {
  const [isChecked, setIsChecked] = useState(false);


  return (
    <form>
      <label>
        Accept Terms:
        <input type="checkbox" checked={isChecked} onChange={() => setIsChecked(!isChecked)} />
      </label>
    </form>
  );
}

export default CheckboxForm;
```

5.2 Handling Radio Buttons

Radio buttons **allow only one selection** from a group.

Example: Handling Radio Buttons in React

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


function RadioButtonForm() {
  const [gender, setGender] = useState("");


  return (
    <form>
      <label>
        Male:
        <input type="radio" name="gender" value="Male"
onChange={(e) => setGender(e.target.value)} />
      </label>
      <label>
        Female:
        <input type="radio" name="gender" value="Female"
onChange={(e) => setGender(e.target.value)} />
      </label>
    </form>
  );
}
```

```
export default RadioButtonForm;
```

5.3 Handling Dropdowns

Dropdowns use select and option elements.

Example: Handling Dropdowns in React

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


function DropdownForm() {
  const [selectedOption, setSelectedOption] = useState("React");


  return (
    <form>
      <label>
        Select Course:
        <select value={selectedOption} onChange={(e) =>
setSelectedOption(e.target.value)}>
          <option value="React">React</option>
          <option value="Angular">Angular</option>
          <option value="Vue">Vue</option>
        </select>
      </label>
```

```
      </form>
    );
}

export default DropdownForm;
```

Chapter 6: Exercise on Events & Forms

1. Create a **login form** with username and password fields.
2. Develop a **newsletter signup form** with checkboxes for subscription preferences.
3. Build a **survey form** using radio buttons and dropdowns.

Conclusion

Mastering **event handling and forms** in React is crucial for building **dynamic and interactive web applications**. Controlled components **ensure data consistency**, while event handling makes applications **responsive to user interactions**.

🚀 **Next Steps:**

- Explore **Formik** and **React Hook Form** for advanced form handling.
- Implement **validation with Yup** to ensure data integrity.
- Build a **real-world login system with authentication**!

REACT.JS COMPONENT LIFECYCLE

CHAPTER 1: INTRODUCTION TO REACT COMPONENT LIFECYCLE

1.1 What is the Component Lifecycle?

In React.js, **component lifecycle** refers to the different **phases a component goes through** from its creation to its removal from the UI. React components have a **lifecycle** that consists of **mounting, updating, and unmounting** phases.

React provides **lifecycle methods** (in class components) and **React Hooks** (in functional components) to execute code at specific stages of a component's existence.

1.2 Why is Understanding the Component Lifecycle Important?

✅ Helps in **optimizing performance** by executing code at the right time.
✅ Allows **fetching data from APIs** at the correct stage.
✅ Helps in **managing side effects,** such as subscriptions or timers.
✅ Prevents **memory leaks** by cleaning up resources when a component unmounts.

1.3 Industry Use Cases

- **Fetching user data** when a component is mounted.
- **Updating UI dynamically** when state or props change.
- **Unsubscribing from APIs or clearing timers** when a component unmounts.

CHAPTER 2: REACT COMPONENT LIFECYCLE PHASES

React components go through **three main phases**:

Mounting Phase (Component is created and added to the DOM).
Updating Phase (Component re-renders due to changes in props or state).
Unmounting Phase (Component is removed from the DOM).

CHAPTER 3: LIFECYCLE METHODS IN CLASS COMPONENTS

3.1 Mounting Phase (Component Creation & DOM Insertion)

These methods are executed when a component is **created and inserted into the DOM**.

Lifecycle Methods in the Mounting Phase

Method	Description	Example Usage
constructor()	Initializes state and binds event handlers	Setting up initial state
static getDerivedStateFromProps(props, state)	Syncs state with props (rarely used)	Update state based on props
render()	Renders the UI	Displaying JSX elements

componentDidMount()	Runs after component is mounted in the DOM	Fetching API data, setting up event listeners

Example: Mounting Phase

```
import React, { Component } from "react";


class UserProfile extends Component {
  constructor(props) {
    super(props);
    this.state = { name: "John Doe" };
    console.log("Constructor: Component is initializing");
  }


  componentDidMount() {
    console.log("ComponentDidMount: Component is mounted to the DOM");
  }


  render() {
    console.log("Render: Rendering component");
    return <h1>Welcome, {this.state.name}!</h1>;
```

```
  }
}
```

export default UserProfile;

✅ componentDidMount() is often used for **fetching API data** after the component is rendered.

3.2 Updating Phase (Re-rendering the Component on State/Props Change)

This phase occurs **whenever a component updates due to changes in state or props**.

Lifecycle Methods in the Updating Phase

Method	Description	Example Usage
static getDerivedStateFromProps(props, state)	Syncs state with props before re-rendering	Update state if props change
shouldComponentUpdate(nextProps, nextState)	Determines if the component should re-render	Optimize performance
render()	Re-renders the UI	Display JSX

getSnapshotBeforeUpdate(prevProps, prevState)	Captures values before DOM updates	Scroll position before update
componentDidUpdate(prevProps, prevState)	Runs after component updates	Fetch updated API data

Example: Updating Phase

```
import React, { Component } from "react";


class Counter extends Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }

  increment = () => {
    this.setState({ count: this.state.count + 1 });
  };

  shouldComponentUpdate(nextProps, nextState) {
```

```
    console.log("ShouldComponentUpdate: Deciding whether to re-render");
    return true; // Always re-render
  }

  componentDidUpdate(prevProps, prevState) {
    console.log("ComponentDidUpdate: Component updated");
  }

  render() {
    console.log("Render: Re-rendering component");
    return (
      <div>
        <h1>Count: {this.state.count}</h1>
        <button onClick={this.increment}>Increment</button>
      </div>
    );
  }
}

export default Counter;
```

✅ shouldComponentUpdate() helps in **optimizing performance** by preventing unnecessary re-renders.

✅ componentDidUpdate() is useful for **fetching new data when props change**.

3.3 Unmounting Phase (Component Removal from the DOM)

This phase occurs **when a component is removed from the UI**.

Lifecycle Methods in the Unmounting Phase

Method	Description	Example Usage
componentWillUnmount()	Cleans up resources before component removal	Remove event listeners, stop API calls

Example: Unmounting Phase

```
import React, { Component } from "react";


class Timer extends Component {
  constructor(props) {
    super(props);
    this.state = { time: 0 };
  }


  componentDidMount() {
```

```
    this.timer = setInterval(() => {
      this.setState({ time: this.state.time + 1 });
    }, 1000);
  }

  componentWillUnmount() {
    console.log("ComponentWillUnmount: Cleaning up resources");
    clearInterval(this.timer);
  }

  render() {
    return <h1>Timer: {this.state.time} seconds</h1>;
  }
}

export default Timer;
```

✅ componentWillUnmount() is **important for cleaning up event listeners, timers, and API calls** to **prevent memory leaks**.

CHAPTER 4: COMPONENT LIFECYCLE IN FUNCTIONAL COMPONENTS (USING HOOKS)

React **functional components** do not use lifecycle methods directly. Instead, they use **React Hooks (useEffect)**.

4.1 Using useEffect() Hook to Handle Lifecycle Events

The useEffect() hook in functional components replaces:

- componentDidMount()
- componentDidUpdate()
- componentWillUnmount()

Example of useEffect() for Mounting and Updating

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


function Timer() {
  const [time, setTime] = useState(0);


  useEffect(() => {
    console.log("useEffect: Component Mounted or Updated");
    const timer = setInterval(() => {
      setTime(prevTime => prevTime + 1);
    }, 1000);


    return () => {
      console.log("Cleanup: Component Unmounting");
```

```
        clearInterval(timer);
      };
  }, []);

  return <h1>Timer: {time} seconds</h1>;
}

export default Timer;
```

✅ **Empty dependency array []** → Runs useEffect() only on mount.
✅ **Returning a cleanup function** → Equivalent to componentWillUnmount().

CHAPTER 5: HANDS-ON EXERCISE

✅ **Create a React Counter Component** using Class and Functional components.
✅ **Modify useEffect()** to fetch data from an API when the component mounts.
✅ **Use componentWillUnmount() or cleanup functions in useEffect()** to stop an API call when the component is removed.

CONCLUSION

Understanding the **React component lifecycle** helps developers:

- **Optimize performance** by controlling re-renders.

- **Manage API calls and subscriptions** efficiently.
- **Ensure proper cleanup** to prevent memory leaks.

🚀 **Next Steps:**

- Learn **React Router** for navigation.
- Explore **State Management (Context API, Redux)**.
- Build a **real-world React project**!

By mastering the **component lifecycle,** you can build **scalable and high-performance React applications**! 🎉🚀

React.js Routing & State Management: React Router (Dynamic & Nested Routes)

Chapter 1: Introduction to React Router

1.1 What is React Router?

React Router is a popular **library for handling navigation** in React applications. Unlike traditional websites where each page reloads upon navigation, React uses **single-page application (SPA)** architecture. This means the **URL changes without reloading the entire page**, ensuring a **smooth user experience**.

1.2 Why Use React Router?

✅ **SPA-Friendly Navigation** – No full-page reloads.
✅ **Dynamic Routing** – Load components based on the URL.
✅ **Nested Routes** – Manage complex layouts efficiently.
✅ **Programmatic Navigation** – Redirect users dynamically.

React Router supports different types of routing:

- **Static Routing** – Defined at the start and doesn't change dynamically.
- **Dynamic Routing** – Generated based on user interaction or data.

💡 **Installation**

Before using React Router, install it in your React project:

```
npm install react-router-dom
```

CHAPTER 2: SETTING UP REACT ROUTER

2.1 Basic Routing Example

To set up routing in React, use **BrowserRouter**, **Routes**, and **Route** components.

Example: Basic Routing

```
import React from "react";

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


function Home() {
  return <h2>Home Page</h2>;
}


function About() {
  return <h2>About Page</h2>;
}


function App() {
  return (
    <Router>
      <Routes>
```

```
            <Route path="/" element={<Home />} />
            <Route path="/about" element={<About />} />
          </Routes>
        </Router>
    );
}

export default App;
```

💡 **Explanation**:

- BrowserRouter – Enables routing.
- Routes – Wraps multiple Route components.
- Route – Defines individual routes (path specifies the URL).
- element – Specifies the component to render.

Chapter 3: Dynamic Routing in React Router

3.1 What is Dynamic Routing?

Dynamic routing enables **loading components dynamically** based on **parameters in the URL**. This is useful for **user profiles, blog posts, and product pages** where content changes based on user input.

3.2 Example: Dynamic Routes Using URL Parameters

The :id syntax in Route defines a **dynamic segment**.

Example: User Profile with Dynamic Route

```
import React from "react";

import { BrowserRouter as Router, Routes, Route, useParams } from "react-router-dom";


function UserProfile() {
  let { id } = useParams();  // Extracts the dynamic value from the URL
  return <h2>Welcome, User {id}</h2>;
}


function App() {
  return (
    <Router>
      <Routes>
        <Route path="/user/:id" element={<UserProfile />} />
      </Routes>
    </Router>
  );
}


export default App;
```

💡 **Explanation**:

- :id in path="/user/:id" makes id a **dynamic parameter**.
- useParams() extracts id from the URL.
- Navigating to /user/5 renders **"Welcome, User 5"**.

3.3 Industry Use Cases for Dynamic Routing

✅ **E-commerce** – Each product has a unique URL (/product/:id).
✅ **Blogs** – Each article has a separate page (/post/:slug).
✅ **Social Media** – User profiles have dynamic URLs (/profile/:username).

CHAPTER 4: NESTED ROUTES IN REACT ROUTER

4.1 What Are Nested Routes?

Nested routes allow **components to be displayed within other components**, making layouts more structured. This is useful when you have **sections inside pages** (e.g., profile settings, dashboards).

4.2 Example: Nested Routing in a Dashboard

```
import React from "react";

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


function Dashboard() {

  return (

    <div>
```

```
        <h2>Dashboard</h2>
        <nav>
          <Link to="profile">Profile</Link> |
          <Link to="settings">Settings</Link>
        </nav>
        <Outlet />  {/* Placeholder for nested routes */}
      </div>
    );
}

function Profile() {
    return <h3>User Profile</h3>;
}

function Settings() {
    return <h3>Account Settings</h3>;
}

function App() {
    return (
      <Router>
```

```
        <Routes>
          <Route path="dashboard" element={<Dashboard />}>
            <Route path="profile" element={<Profile />} />
            <Route path="settings" element={<Settings />} />
          </Route>
        </Routes>
      </Router>
    );
}

export default App;
```

💡 **Explanation**:

- Outlet acts as a **placeholder** for nested components.
- /dashboard loads the Dashboard component.
- /dashboard/profile loads Profile.
- /dashboard/settings loads Settings.

4.3 Real-World Use Cases of Nested Routing

✅ **Admin Dashboards** – /dashboard/analytics, /dashboard/users.
✅ **User Profiles** – /profile/info, /profile/edit.
✅ **E-commerce** – /products/:id/reviews, /products/:id/details.

CHAPTER 5: NAVIGATION WITH LINK & USENAVIGATE

5.1 Using <Link> for Navigation

Instead of using <a> tags (which reload the page), use React Router's <Link> component.

Example: Navigating Using <Link>

```
import { Link } from "react-router-dom";


function Navbar() {
  return (
    <nav>
      <Link to="/">Home</Link> |
      <Link to="/about">About</Link>
    </nav>
  );
}
```

💡 **Why use <Link> instead of <a>?**

- Prevents **full page reload**.
- Maintains **SPA behavior**.
- Faster navigation.

5.2 Using useNavigate for Programmatic Navigation

useNavigate() allows navigation **inside functions**, useful for redirects after login/logout.

Example: Redirecting After Button Click

```
import { useNavigate } from "react-router-dom";


function Home() {
  const navigate = useNavigate();


  const goToDashboard = () => {
    navigate("/dashboard");
  };


  return <button onClick={goToDashboard}>Go to
Dashboard</button>;
}
```

Chapter 6: Exercises on React Router

Exercise 1: Create a Multi-Page App

1. Set up React Router with /home, /about, and /contact pages.
2. Add navigation links using <Link>.

Exercise 2: Implement Dynamic Routes

1. Create a ProductDetail page with URL /product/:id.
2. Extract id using useParams().

Exercise 3: Build a Nested Route Structure

1. Create a Dashboard component with Profile and Settings as nested routes.
2. Use <Outlet> to display nested components.

CONCLUSION

React Router is essential for **building SPAs**, allowing seamless navigation without page reloads. By mastering **dynamic and nested routes**, developers can create **structured, user-friendly applications**.

🚀 Next Steps:

- Learn about **protected routes** for authentication.
- Explore **React Router transitions** for smooth navigation effects.
- Implement **lazy loading for performance optimization**!

USESTATE AND USEEFFECT HOOKS IN REACT

CHAPTER 1: INTRODUCTION TO REACT HOOKS

1.1 What are React Hooks?

Hooks are special functions that **allow functional components to use state and lifecycle features** in React. Before Hooks, only **class components** could manage state and lifecycle methods.

React introduced Hooks in **version 16.8**, enabling functional components to be **stateful and dynamic**.

1.2 Why Use Hooks?

✅ **Simplifies Code** – No need for complex class components.
✅ **Better Code Reusability** – Reusable functions instead of class-based logic.
✅ **Easier State Management** – useState replaces this.state.
✅ **Better Lifecycle Management** – useEffect replaces lifecycle methods.

1.3 Common React Hooks

Hook	Purpose
useState	Manages state in functional components.
useEffect	Handles side effects like API calls and event listeners.
useContext	Provides global state management without prop drilling.

useRef	Works with DOM elements or persists values across renders.
useReducer	An alternative to useState for complex state logic.
useMemo	Optimizes performance by caching values.

CHAPTER 2: USESTATE HOOK – MANAGING STATE IN FUNCTIONAL COMPONENTS

2.1 What is useState?

The useState hook **allows functional components to have state**. It replaces the need for this.state in class components.

2.2 Syntax of useState

const [state, setState] = useState(initialValue);

- state: Holds the current value.
- setState: Updates the state.
- initialValue: The starting value for the state.

2.3 Example: Simple Counter using useState

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


function Counter() {
  const [count, setCount] = useState(0);
```

```
  return (
    <div>
      <h1>Count: {count}</h1>
      <button onClick={() => setCount(count + 1)}>Increment</button>
      <button onClick={() => setCount(count - 1)}>Decrement</button>
    </div>
  );
}

export default Counter;
```

✅ setCount(count + 1) updates the state and triggers a re-render.

2.4 Updating State with Previous State

```
setCount(prevCount => prevCount + 1);
```

Using the previous state ensures the correct value, especially in **async updates**.

2.5 Using useState with Objects

```
const [user, setUser] = useState({ name: "Alice", age: 25 });
```

```
function updateAge() {
  setUser(prevUser => ({ ...prevUser, age: prevUser.age + 1 }));
}
```

✅ **Spread operator (...prevUser)** keeps existing properties while updating only age.

2.6 Handling Forms with useState

```
function Form() {
  const [name, setName] = useState("");

  return (
    <div>
      <input type="text" value={name} onChange={e => setName(e.target.value)} />
      <p>Hello, {name}!</p>
    </div>
  );
}
```

✅ Updates input value dynamically using onChange.

Chapter 3: useEffect Hook – Handling Side Effects

3.1 What is useEffect?

The useEffect hook **performs side effects** in functional components.

Side effects include:

✅ Fetching data from APIs
✅ Updating the DOM
✅ Subscribing to events
✅ Setting up timers

3.2 Syntax of useEffect

```
useEffect(() => {
  // Side effect logic here
}, [dependencies]);
```

- **Function (()=>{})** – Runs the side effect.
- **Dependencies ([])** – Controls when the effect runs.

3.3 Example: Fetching API Data with useEffect

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

function FetchData() {
  const [users, setUsers] = useState([]);

  useEffect(() => {
    fetch("https://jsonplaceholder.typicode.com/users")
      .then(response => response.json())
```

```
        .then(data => setUsers(data));
  }, []);

  return (
    <ul>
      {users.map(user => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  );
}

export default FetchData;
```

✅ **Runs once ([] as dependency)** when the component mounts.

3.4 When Does useEffect Run?

Dependency	Runs When
useEffect(()=>{ }, [])	Runs **only once** (on mount).
useEffect(()=>{ }, [count])	Runs **when count changes**.
useEffect(()=>{ })	Runs **on every render**.

3.5 Example: Updating the Title Dynamically

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


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


  useEffect(() => {
    document.title = `Count: ${count}`;
  }, [count]);


  return (
    <button onClick={() => setCount(count + 1)}>Increment</button>
  );
}
```

✅ **Document title updates** whenever count changes.

3.6 Cleaning Up Side Effects (Unmounting Phase)

To avoid memory leaks, **clean up side effects** when a component unmounts.

Example: Cleaning Up an Interval Timer

```
useEffect(() => {
```

```
  const timer = setInterval(() => {
    console.log("Timer running...");
  }, 1000);

  return () => {
    clearInterval(timer); // Cleanup function
  };
}, []);
```

✅ **Returns a cleanup function** that runs before the component unmounts.

Chapter 4: Hands-on Practice & Assignments

4.1 Hands-on Exercises

✅ **Create a Counter App** using useState.
✅ **Fetch API data** using useEffect.
✅ **Build a Timer App** with useEffect and cleanup function.

4.2 Assignment: Build a To-Do List App with useState and useEffect

Requirements:

Use useState to manage a **list of tasks**.
Use useEffect to **save tasks in local storage**.
Fetch tasks from **local storage when the app loads**.

Expected Output:

- Add new tasks.
- Remove tasks.
- Persist tasks even after a page refresh.

Solution (ToDo.js)

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


function ToDoList() {
  const [tasks, setTasks] = useState([]);


  useEffect(() => {
    const storedTasks = JSON.parse(localStorage.getItem("tasks"));
    if (storedTasks) setTasks(storedTasks);
  }, []);


  useEffect(() => {
    localStorage.setItem("tasks", JSON.stringify(tasks));
  }, [tasks]);


  function addTask() {
    setTasks([...tasks, `Task ${tasks.length + 1}`]);
```

```
  }

  return (
    <div>
      <button onClick={addTask}>Add Task</button>
      <ul>
        {tasks.map((task, index) => (
          <li key={index}>{task}</li>
        ))}
      </ul>
    </div>
  );
}

export default ToDoList;
```

CONCLUSION

Key Takeaways

✅ **useState** manages **state** in functional components.
✅ **useEffect** handles **side effects, API calls, and event listeners**.
✅ **Cleanup functions prevent memory leaks** in useEffect.

🚀 Next Steps:

- Learn **React Router** for navigation.
- Explore **useContext and Redux** for state management.
- Build a **real-world project using Hooks**!

By mastering useState and useEffect, you can **develop scalable React applications** with dynamic features! 🎉🚀

USEREDUCER HOOK

CHAPTER 1: INTRODUCTION TO STATE MANAGEMENT IN REACT

1.1 Understanding State Management in React

React applications often require **state management** to handle user interactions, API data, and UI updates. For small applications, **useState** is sufficient, but as the app grows, managing **global state** across multiple components becomes challenging.

React provides **Context API** and **useReducer** as alternatives to third-party state management libraries like Redux.

1.2 Why Use Context API and useReducer?

✅ **Avoid Prop Drilling** – No need to pass props down multiple components.
✅ **Centralized State Management** – Allows shared state across components.
✅ **Improves Code Maintainability** – Reduces unnecessary prop-passing.
✅ **Better Performance** – Avoids unnecessary re-renders.

1.3 Industry Use Cases of Context API & useReducer

🚀 **Authentication System** – Store user authentication state (login/logout).
🚀 **E-commerce Cart System** – Manage cart items and pricing dynamically.
🚀 **Theme Management** – Enable light/dark mode toggling across the app.
🚀 **Global Notifications** – Manage app-wide alerts and messages.

CHAPTER 2: REACT CONTEXT API

2.1 What is Context API?

The **React Context API** is a built-in feature that allows **global state management** without prop drilling. It provides a way to **share data** between components at different levels of the component tree.

2.2 How Does Context API Work?

Context API works with **three key components**:

1. **React.createContext()** – Creates a context object.
2. **Provider Component** – Supplies state to child components.
3. **Consumer or useContext() Hook** – Accesses context data inside components.

2.3 Implementing Context API in React

Step 1: Create Context (ThemeContext.js)

```
import React, { createContext, useState } from "react";


// Create Context
export const ThemeContext = createContext();


// Provider Component
export const ThemeProvider = ({ children }) => {
  const [theme, setTheme] = useState("light");
```

```
 // Function to toggle theme
 const toggleTheme = () => {
  setTheme(theme === "light" ? "dark" : "light");
 };

 return (
  <ThemeContext.Provider value={{ theme, toggleTheme }}>
   {children}
  </ThemeContext.Provider>
 );
};
```

Step 2: Use Context in a Component (ThemeSwitcher.js)

```
import React, { useContext } from "react";
import { ThemeContext } from "./ThemeContext";

function ThemeSwitcher() {
 const { theme, toggleTheme } = useContext(ThemeContext);

 return (
```

```
  <div style={{ background: theme === "light" ? "#fff" : "#333", color: theme === "light" ? "#000" : "#fff", padding: "20px", textAlign: "center" }}>
    <h1>{theme.toUpperCase()} MODE</h1>
    <button onClick={toggleTheme}>Toggle Theme</button>
  </div>
 );
}

export default ThemeSwitcher;
```

Step 3: Wrap Components with Provider (App.js)

```
import React from "react";
import { ThemeProvider } from "./ThemeContext";
import ThemeSwitcher from "./ThemeSwitcher";

function App() {
 return (
  <ThemeProvider>
   <ThemeSwitcher />
  </ThemeProvider>
 );
```

}

export default App;

2.4 Benefits of Using Context API

✅ **Eliminates prop drilling** – No need to pass state manually to each component.
✅ **Easier to manage global state** – Ideal for themes, authentication, and language settings.
✅ **Better readability and modularity** – Keeps code **clean and reusable**.

2.5 Exercise on Context API

1. **Modify the above example** to include a language switcher (English/Spanish).
2. **Create a user authentication context** to store login/logout state.
3. **Build a shopping cart context** to manage cart items globally.

CHAPTER 3: USEREDUCER HOOK IN REACT

3.1 What is useReducer?

The **useReducer** hook is an alternative to **useState**, used for managing **complex state logic** in React applications.

3.2 When to Use useReducer?

✅ When **state logic is complex** and involves multiple sub-values.
✅ When **multiple state updates** are needed based on user actions.
✅ When using **Redux-like state management** inside components.

3.3 How Does useReducer Work?

useReducer consists of **three components**:

1. **State** – The initial state value.
2. **Reducer Function** – Defines how the state should change based on actions.
3. **Dispatch Function** – Triggers state changes by sending an action.

3.4 Implementing useReducer in React

Step 1: Define Reducer Function (counterReducer.js)

```
const counterReducer = (state, action) => {
 switch (action.type) {
  case "INCREMENT":
   return { count: state.count + 1 };
  case "DECREMENT":
   return { count: state.count - 1 };
  case "RESET":
   return { count: 0 };
  default:
```

```
    return state;
  }
};

export default counterReducer;
```

Step 2: Use useReducer Hook in a Component (Counter.js)

```
import React, { useReducer } from "react";
import counterReducer from "./counterReducer";

function Counter() {
  const [state, dispatch] = useReducer(counterReducer, { count: 0 });

  return (
    <div style={{ textAlign: "center", padding: "20px" }}>
      <h1>Count: {state.count}</h1>
      <button onClick={() => dispatch({ type: "INCREMENT" })}>+</button>
      <button onClick={() => dispatch({ type: "DECREMENT" })}>-</button>
      <button onClick={() => dispatch({ type: "RESET" })}>Reset</button>
```

```
    </div>
  );
}

export default Counter;
```

Step 3: Render Counter Component (App.js)

```
import React from "react";
import Counter from "./Counter";

function App() {
  return (
    <div>
      <Counter />
    </div>
  );
}

export default App;
```

3.5 Advantages of useReducer Over useState

Feature	useState	useReducer
Best for	Simple state updates	Complex state logic
State Management	Uses single values	Uses an object with multiple values
Performance	Can cause unnecessary re-renders	Optimized for multiple updates

3.6 Exercise on useReducer

1. **Modify the Counter example** to include step increments (increase by 5 instead of 1).
2. **Create a to-do list** where useReducer manages adding, deleting, and completing tasks.
3. **Build a form validation system** where useReducer handles form errors and field updates.

CHAPTER 4: COMBINING CONTEXT API & USEREDUCER

4.1 Why Use Context API with useReducer?

For large-scale applications, **Context API** helps provide **global access** to state, while **useReducer** efficiently manages state changes.

4.2 Example: Global State Management with Context API & useReducer

Step 1: Create a Counter Context

```
import React, { createContext, useReducer } from "react";
```

```
const CounterContext = createContext();
const counterReducer = (state, action) => {
 switch (action.type) {
  case "INCREMENT":
   return { count: state.count + 1 };
  case "DECREMENT":
   return { count: state.count - 1 };
  default:
   return state;
 }
};

export const CounterProvider = ({ children }) => {
 const [state, dispatch] = useReducer(counterReducer, { count: 0 });

 return (
  <CounterContext.Provider value={{ state, dispatch }}>
   {children}
  </CounterContext.Provider>
 );
```

```
};


export default CounterContext;
```

CONCLUSION

By learning **Context API and useReducer,** developers can efficiently manage **global state** and handle **complex state logic** in React applications.

🚀 **Next Steps:**
✓ Use **useReducer for form validation**.
✓ Apply **Context API to user authentication systems**.
✓ Build a **global store using Context API & useReducer**.

Mastering these concepts will take your React skills to the **next level!** 🚀 🔥

STYLING IN REACT: CSS MODULES

CHAPTER 1: INTRODUCTION TO CSS MODULES

1.1 What are CSS Modules?

CSS Modules are a **scoped styling solution** in React that prevents **global CSS conflicts** by **generating unique class names** for each component. Unlike traditional CSS, where styles are **globally applied**, CSS Modules **encapsulate styles within a specific component**, ensuring that styles do not leak into other parts of the application.

1.2 Why Use CSS Modules in React?

✅ **Scoped Styles** – Styles apply only to the component where they are imported.
✅ **Avoid Naming Conflicts** – Automatically generates unique class names.
✅ **Better Maintainability** – Makes components more reusable and modular.
✅ **Works with All CSS Features** – Supports variables, media queries, animations, etc.

CHAPTER 2: SETTING UP CSS MODULES IN REACT

2.1 Creating a CSS Module

By convention, CSS Module files should have the .module.css extension.

Steps to Create a CSS Module in React:

1. Create a CSS file named Button.module.css.
2. Define styles inside the CSS module.
3. Import and use the styles in a React component.

Example: Creating and Using a CSS Module

Step 1: Define Styles in a CSS Module (Button.module.css)

```
.button {
  background-color: #007bff;
  color: white;
  padding: 10px 20px;
  border: none;
  border-radius: 5px;
  font-size: 16px;
  cursor: pointer;
}

.button:hover {
  background-color: #0056b3;
}
```

Step 2: Import the CSS Module in a React Component (Button.js)

```
import React from "react";
```

```
import styles from "./Button.module.css";  // Importing the CSS module


function Button() {
  return <button className={styles.button}>Click Me</button>;
}


export default Button;
```

💡 **Key Takeaways:**
✅ import styles from "./Button.module.css"; – Imports styles as a JavaScript object.
✅ styles.button – Accesses the .button class from Button.module.css.
✅ Class names are **scoped** and will not interfere with other components.

CHAPTER 3: HOW CSS MODULES WORK UNDER THE HOOD

3.1 Automatic Class Name Generation

When using CSS Modules, React **renames class names dynamically** to avoid conflicts.
For example, if the original CSS file has:

```
.button {
  background-color: blue;
}
```

React may transform it into:

```
.Button_button__2GkIe {
  background-color: blue;
}
```

This ensures that the class name is **unique** and does not override other styles.

3.2 Example of Encapsulation in CSS Modules

If you create another CSS file Header.module.css:

```
.button {
  background-color: red;
}
```

And import it inside Header.js:

```
import styles from "./Header.module.css";
```

It **will not** override styles in Button.module.css due to React's unique **scoped class names**.

CHAPTER 4: STYLING MULTIPLE ELEMENTS WITH CSS MODULES

4.1 Assigning Multiple Classes

Sometimes, you may want to apply multiple styles to an element.

Example: Combining Multiple Classes

```
.primary {
  background-color: green;
```

```
  color: white;
}

.large {
  padding: 15px 30px;
  font-size: 18px;
}

<button className={`${styles.primary} ${styles.large}`}>Click Me</button>
```

💡 **Tip:** Use template literals (${styles.class1} ${styles.class2}) to combine classes dynamically.

CHAPTER 5: USING CSS MODULES WITH DYNAMIC STYLING

5.1 Applying Styles Conditionally

Use JavaScript logic to dynamically change styles based on component state.

Example: Conditional Styling with CSS Modules

```
.active {
  background-color: limegreen;
}

.inactive {
```

```
  background-color: gray;
}
import React, { useState } from "react";
import styles from "./Toggle.module.css";

function ToggleButton() {
  const [isActive, setIsActive] = useState(false);

  return (
    <button
      className={isActive ? styles.active : styles.inactive}
      onClick={() => setIsActive(!isActive)}
    >
      {isActive ? "Active" : "Inactive"}
    </button>
  );
}

export default ToggleButton;
```

💡 **Explanation:**

- useState(false) tracks the active state.

- The button switches between active and inactive styles dynamically.

Chapter 6: Using CSS Modules with Media Queries

CSS Modules **fully support media queries** for responsive designs.

Example: Responsive Button in CSS Module

```
.button {
  padding: 10px 20px;
  font-size: 16px;
}

@media (max-width: 600px) {
  .button {
    font-size: 12px;
    padding: 8px 16px;
  }
}
```

This ensures the button adjusts its **size on smaller screens**.

Chapter 7: Best Practices for CSS Modules in React

✅ **Use Descriptive Class Names** – Avoid generic names like .container or .box.

✅ **Group Component-Specific Styles** – Keep related styles in one module file.

✅ **Use Variables for Colors & Fonts** – Maintain consistency with CSS variables.

✅ **Avoid Deep Nesting** – Keep styles modular and easy to manage.

✅ **Separate Global Styles** – Use CSS Modules for components but keep **global styles** in a standard CSS file (e.g., index.css).

Chapter 8: Case Study – Using CSS Modules in a Large-Scale Project

Problem

An e-commerce company faced **styling conflicts** across different product pages. Developers had difficulty maintaining styles **without overriding existing classes**.

Solution

They migrated to **CSS Modules** to:

1. **Scope styles** per component (ProductCard.module.css, Cart.module.css).
2. **Reduce CSS conflicts** using unique class names.
3. **Improve maintainability** with reusable components.

Outcome

✅ **Reduced CSS conflicts by 80%**.
✅ **Faster UI development** with isolated styles.
✅ **Easier debugging** by tracking styles within their components.

CHAPTER 9: EXERCISES ON CSS MODULES

Exercise 1: Create a Styled Button

1. Create a CSS Module named Button.module.css.
2. Define styles for .primary-button (blue background) and .secondary-button (gray background).
3. Import and use them in a Button.js component.

Exercise 2: Build a Styled Card Component

1. Create a **Card.module.css** file with styles for a card layout.
2. Use CSS Modules to style Card.js.
3. Make sure the card is **responsive** using media queries.

Exercise 3: Implement Dynamic Theme Switching

1. Create a toggle button to **switch between dark mode and light mode** using CSS Modules.
2. Use React useState() to apply different styles dynamically.

CONCLUSION

CSS Modules are an **efficient way to scope styles** in React**, prevent CSS conflicts,** and **maintain modularity**. They **enhance scalability** in large projects**,** making it easier to manage styles per component.

🚀 Next Steps:

- Learn **SCSS Modules** for more advanced styling options.
- Explore **CSS-in-JS libraries** like **Styled-Components** for dynamic styling.
- Implement **Theme Switching** with CSS Modules for dark/light mode!

STYLING IN REACT: STYLED-COMPONENTS

CHAPTER 1: INTRODUCTION TO STYLED-COMPONENTS

1.1 What are Styled-components?

Styled-components is a popular **CSS-in-JS** library that enables developers to write **component-level styles** in JavaScript. It allows you to create **fully encapsulated, reusable styled components** without needing separate CSS files.

Styled-components use **template literals** to define styles, making them more readable and modular.

💡 **Installation of Styled-components:**
To use Styled-components in React, install the package using npm or yarn:

npm install styled-components

or

yarn add styled-components

1.2 Why Use Styled-components?

✅ **Scoped Styling** – No class name conflicts (styles apply only to specific components).
✅ **Dynamic Styling** – Apply styles based on **props or state**.
✅ **Better Code Maintainability** – Write CSS directly inside

components.

✅ **Supports Theming** – Easily switch themes in your app.

CHAPTER 2: CREATING AND USING STYLED-COMPONENTS

2.1 Basic Syntax of Styled-components

Styled-components use the styled object to define components with styles.

Example: Creating a Styled Button

```
import React from "react";

import styled from "styled-components";


// Styled button component

const Button = styled.button`

  background-color: blue;

  color: white;

  padding: 10px 20px;

  font-size: 18px;

  border: none;

  border-radius: 5px;

  cursor: pointer;


  &:hover {
```

```
    background-color: darkblue;
  }
`;

function App() {
  return <Button>Click Me</Button>;
}

export default App;
```

✅ **Styled-components allow defining styles directly in the component.**
✅ **CSS inside template literals maintains proper syntax.**

2.2 Using Props for Dynamic Styling

You can pass **props** to styled-components to create **dynamic styles**.

Example: Changing Button Color Based on Props

```
import React from "react";
import styled from "styled-components";

const Button = styled.button`
  background-color: ${(props) => (props.primary ? "blue" : "gray")};
  color: white;
```

```
  padding: 10px 20px;
  font-size: 18px;
  border: none;
  border-radius: 5px;
  cursor: pointer;

  &:hover {
    background-color: ${(props) => (props.primary ? "darkblue" : "darkgray")};
  }
`;

function App() {
  return (
    <div>
      <Button primary>Primary Button</Button>
      <Button>Default Button</Button>
    </div>
  );
}
```

```
export default App;
```

✅ **Props determine whether the button is primary (blue) or default (gray).**
✅ **Props are dynamically applied to modify CSS styles.**

CHAPTER 3: THEMING WITH STYLED-COMPONENTS

3.1 Why Use Themes?

Themes allow developers to define **global design styles** (colors, fonts, spacing) and apply them throughout the application. Styled-components **ThemeProvider** makes it easy to manage themes.

3.2 Example: Creating a Theme in React

```
import React from "react";
import styled, { ThemeProvider } from "styled-components";

// Define a theme
const theme = {
  primaryColor: "blue",
  secondaryColor: "gray",
  textColor: "white"
};

// Styled button with theme props
```

```
const Button = styled.button`
  background-color: ${(props) => props.theme.primaryColor};
  color: ${(props) => props.theme.textColor};
  padding: 10px 20px;
  font-size: 18px;
  border: none;
  border-radius: 5px;
  cursor: pointer;
`;

function App() {
  return (
    <ThemeProvider theme={theme}>
      <Button>Themed Button</Button>
    </ThemeProvider>
  );
}

export default App;
```

✅ **The ThemeProvider allows centralized theme management.**
✅ **Theme values are accessed using props.theme.**

CHAPTER 4: EXTENDING AND ANIMATING STYLED-COMPONENTS

4.1 Extending Styled-components

You can extend existing styled-components to reuse styles.

Example: Creating a Secondary Button by Extending the Primary Button

```
const PrimaryButton = styled.button`
  background-color: blue;
  color: white;
  padding: 10px 20px;
  border: none;
  border-radius: 5px;
`;

const SecondaryButton = styled(PrimaryButton)`
  background-color: gray;
`;
```

✅ **Saves time by extending existing styles.**
✅ **Reuses styles while adding modifications.**

4.2 Adding Animations with Styled-components

Styled-components supports CSS animations using keyframes.

Example: Animating a Button on Hover

```
import styled, { keyframes } from "styled-components";


// Define a keyframe animation
const fadeIn = keyframes`
  from {
    opacity: 0;
  }
  to {
    opacity: 1;
  }
`;


const AnimatedButton = styled.button`
  background-color: blue;
  color: white;
  padding: 10px 20px;
  font-size: 18px;
  border: none;
  border-radius: 5px;
  cursor: pointer;
```

```
  animation: ${fadeIn} 2s ease-in-out;

  &:hover {
    background-color: darkblue;
  }
`;

function App() {
  return <AnimatedButton>Hover Me</AnimatedButton>;
}

export default App;
```

✅ **Creates smooth animations with keyframes.**
✅ **Uses animation property inside styled-components.**

CHAPTER 5: BEST PRACTICES FOR USING STYLED-COMPONENTS

5.1 When to Use Styled-components?

✓ For component-scoped styles.
✓ When you need dynamic styling.
✓ For theme-based designs.
✓ For easily maintainable and reusable styles.

5.2 When to Avoid Styled-components?

❌ **For global styles that don't change dynamically.**
❌ **If performance is a concern (Styled-components generate unique class names dynamically).**

Chapter 6: Hands-on Practice & Exercises

Exercise 1: Create a Styled Button Component

✅ Define a **styled button** with hover effects.
✅ Use **props** to change colors dynamically.

Exercise 2: Build a Theme Switcher

✅ Create **two themes (light & dark mode)**.
✅ Implement a **toggle button** to switch between themes.

Exercise 3: Animate a Loading Spinner

✅ Use **Styled-components keyframes** to create a rotating spinner.

Conclusion

Styled-components provide **scoped, reusable, and dynamic styles** in React applications. With **theme support, animations, and component-based styling**, Styled-components improve the **readability and maintainability of CSS in React projects**.

🚀 **Next Steps:**

- Explore **global styles** with Styled-components.
- Learn about **CSS-in-JS performance optimizations**.
- Implement **Styled-components in a real-world project**!

By mastering Styled-components, you can **build modern, scalable, and well-structured React applications with maintainable styles**!

STUDY MATERIAL: STYLING IN REACT WITH TAILWIND CSS

CHAPTER 1: INTRODUCTION TO TAILWIND CSS

1.1 What is Tailwind CSS?

Tailwind CSS is a **utility-first CSS framework** that provides a set of **predefined classes** to style web applications directly in HTML or JSX. Unlike traditional CSS frameworks (e.g., Bootstrap), which provide pre-designed components, Tailwind allows developers to **customize designs rapidly without writing custom CSS**.

1.2 Key Features of Tailwind CSS

✅ **Utility-First Approach** – Provides small, reusable CSS classes for styling.
✅ **Highly Customizable** – Modify themes using the tailwind.config.js file.
✅ **Responsive Design** – Built-in breakpoints for mobile-friendly designs.
✅ **Faster Development** – No need to write separate CSS files; styles are directly applied using class names.
✅ **Performance Optimized** – Removes unused styles with **tree-shaking** in production builds.

1.3 Industry Use Cases of Tailwind CSS

🚀 **E-commerce Websites** – Custom product grids and user interfaces.
🚀 **Dashboard & Admin Panels** – Rapid UI development for analytics and data visualization.
🚀 **Portfolio Websites** – Easily customizable layouts and designs.

🚀 **Startup MVPs** – Quick prototyping without complex CSS management.

CHAPTER 2: INSTALLING TAILWIND CSS IN A REACT PROJECT

2.1 Setting Up a React App with Tailwind CSS

To use Tailwind CSS in a React project, follow these steps:

Step 1: Create a React App

Open a terminal and create a new React project:

npx create-react-app my-tailwind-app

cd my-tailwind-app

Step 2: Install Tailwind CSS & Dependencies

Run the following command to install Tailwind and its dependencies:

npm install tailwindcss postcss autoprefixer

Step 3: Generate Tailwind Configuration Files

Initialize Tailwind by running:

npx tailwindcss init -p

This will create two files:

✅ tailwind.config.js (for customizing Tailwind settings)

✅ postcss.config.js (for integrating with PostCSS)

Step 4: Configure Tailwind in Your Project

Open tailwind.config.js and modify the content array to scan your React files:

```
module.exports = {
 content: ["./src/**/*.{js,jsx,ts,tsx}"],
 theme: {
  extend: {},
 },
 plugins: [],
};
```

Step 5: Add Tailwind to Your CSS

In src/index.css, replace the existing content with:

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Step 6: Run the React App

```
npm start
```

Now your React project is set up with Tailwind CSS! 🚀

CHAPTER 3: USING TAILWIND CSS IN REACT COMPONENTS

3.1 Applying Tailwind Classes in JSX

Instead of writing CSS files, you apply Tailwind classes directly in JSX.

Example: Basic Styled Button

```
function App() {
 return (
  <button className="bg-blue-500 text-white px-4 py-2 rounded-md hover:bg-blue-700">
    Click Me
  </button>
 );
}

export default App;
```

✅ bg-blue-500 – Background color
✅ text-white – Text color
✅ px-4 py-2 – Padding (horizontal & vertical)
✅ rounded-md – Rounded corners
✅ hover:bg-blue-700 – Changes background color on hover

3.2 Creating a Responsive Navbar

Tailwind provides **responsive utility classes** like sm:, md:, lg:, and xl:.

```
function Navbar() {
 return (
  <nav className="bg-gray-800 text-white p-4">
```

```
    <div className="container mx-auto flex justify-between items-center">
      <h1 className="text-xl font-bold">My Website</h1>
      <ul className="hidden md:flex space-x-4">
        <li><a href="#" className="hover:text-gray-300">Home</a></li>
        <li><a href="#" className="hover:text-gray-300">About</a></li>
        <li><a href="#" className="hover:text-gray-300">Contact</a></li>
      </ul>
      <button className="md:hidden">☰</button>
    </div>
  </nav>
 );
}

export default Navbar;
```

✅ hidden md:flex – Navbar is hidden on mobile but visible on medium screens.
✅ space-x-4 – Adds horizontal spacing between items.
✅ hover:text-gray-300 – Changes text color on hover.

3.3 Grid Layout with Tailwind CSS

Using **Flexbox (flex) and Grid (grid),** we can quickly create layouts.

Example: Responsive Product Grid

```
function ProductGrid() {
 return (
  <div className="grid grid-cols-1 sm:grid-cols-2 md:grid-cols-3 gap-6 p-6">
   <div className="bg-white shadow-md p-4 rounded-md">Product 1</div>
   <div className="bg-white shadow-md p-4 rounded-md">Product 2</div>
   <div className="bg-white shadow-md p-4 rounded-md">Product 3</div>
  </div>
 );
}

export default ProductGrid;
```

✅ grid-cols-1 sm:grid-cols-2 md:grid-cols-3 – Defines different column structures for different screen sizes.
✅ gap-6 – Adds spacing between items.
✅ shadow-md – Adds a shadow effect to cards.

CHAPTER 4: CUSTOMIZING TAILWIND CSS

4.1 Extending Tailwind Config (tailwind.config.js)

You can **customize themes, colors, and fonts** in Tailwind.

Example: Adding Custom Colors

Modify tailwind.config.js:

```
module.exports = {
 theme: {
  extend: {
   colors: {
    primary: "#1D4ED8", // Custom Blue
    secondary: "#F59E0B", // Custom Orange
   },
  },
 },
};
```

Use these colors in components:

```
<button className="bg-primary text-white px-4 py-2">Primary Button</button>

<button className="bg-secondary text-white px-4 py-2">Secondary Button</button>
```

4.2 Dark Mode in Tailwind CSS

Enable dark mode by adding "darkMode": "class" to tailwind.config.js:

```
module.exports = {
  darkMode: "class",
};
```

Now, apply dark mode styles in JSX:

```
<div className="bg-white dark:bg-gray-900 text-black dark:text-white p-6">
  Dark Mode Example
</div>
```

Toggle dark mode dynamically:

```
document.body.classList.toggle("dark");
```

CHAPTER 5: HANDS-ON EXERCISES

5.1 Exercise 1: Create a Responsive Card Component

✅ Design a **card** component using Tailwind's shadow, rounded, and hover effects.

5.2 Exercise 2: Build a Hero Section

✅ Create a **landing page hero section** with a call-to-action (CTA) button.

5.3 Exercise 3: Implement a Dark Mode Toggle

✅ Create a **toggle button** to switch between light and dark mode.

CONCLUSION

Tailwind CSS is a **powerful, utility-first CSS framework** that simplifies **styling in React applications**. By using predefined classes, developers can **build modern, responsive UIs quickly** without writing custom CSS.

🚀 **Next Steps:**

✓ Explore Tailwind plugins like @tailwindcss/forms and @tailwindcss/typography.

✓ Build a **real-world project** like a portfolio or an e-commerce UI.

✓ Learn **Tailwind with Next.js** for even faster development.

By mastering Tailwind CSS, you'll **boost your React UI development skills** and create **high-quality, responsive designs effortlessly!** 🚀 🔥

HANDS-ON PRACTICE: CREATE A REACT-BASED RESUME WEBSITE

CHAPTER 1: INTRODUCTION TO BUILDING A RESUME WEBSITE WITH REACT

1.1 Why Build a Resume Website?

A **resume website** serves as an **interactive portfolio**, allowing professionals to showcase their **skills, experience, and projects** online. Unlike traditional resumes, a web-based resume provides:

✅ **Better visibility** – Employers can find your work easily.
✅ **Interactivity** – Showcases skills dynamically.
✅ **Customizability** – Add projects, animations, and contact forms.
✅ **Easy updates** – Modify sections without re-creating the entire document.

1.2 What You Will Learn

- Setting up a React project
- Using **React components** for structured layouts
- Styling with **CSS Modules**
- Adding interactivity using **React Hooks**
- Deploying the project on **Netlify or Vercel**

CHAPTER 2: SETTING UP YOUR REACT RESUME WEBSITE

2.1 Prerequisites

- Install **Node.js** (Download from https://nodejs.org/)
- Install **npm** or **yarn** (included with Node.js)
- Basic understanding of **React and JSX**

2.2 Creating a React App

To start a new React project, run the following commands in your terminal:

```
npx create-react-app react-resume

cd react-resume

npm start  # Starts the development server
```

This sets up a React project with a **default folder structure**.

CHAPTER 3: BUILDING THE RESUME STRUCTURE

3.1 Project Folder Structure

```
react-resume/
 ├── public/
 ├── src/
 │    ├── components/
 │    │    ├── Header.js
 │    │    ├── About.js
 │    │    ├── Experience.js
 │    │    ├── Projects.js
 │    │    ├── Contact.js
```

```
│   │   │ ——— Footer.js
│   │ ——— App.js
│   │ ——— index.js
│ ——— package.json
│ ——— README.md
```

Explanation:

- Header.js – Contains **name and navigation links**
- About.js – Displays **profile summary and skills**
- Experience.js – Lists **work experience**
- Projects.js – Showcases **portfolio projects**
- Contact.js – Includes **contact form or social links**
- Footer.js – Displays **copyright and links**

CHAPTER 4: CREATING COMPONENTS FOR RESUME SECTIONS

4.1 Creating the Header Component

File: src/components/Header.js

```
import React from "react";
import styles from "./Header.module.css";  // Importing CSS Module

function Header() {
  return (
```

```
      <header className={styles.header}>
        <h1>John Doe</h1>
        <nav>
          <a href="#about">About</a>
          <a href="#experience">Experience</a>
          <a href="#projects">Projects</a>
          <a href="#contact">Contact</a>
        </nav>
      </header>
    );
}

export default Header;
```

- **Uses CSS Modules** to style the header.
- **Navigation links** scroll to different sections.

4.2 Creating the About Section

File: src/components/About.js

```
import React from "react";
import styles from "./About.module.css";
```

```
function About() {
  return (
    <section id="about" className={styles.about}>
      <h2>About Me</h2>
      <p>I'm a Frontend Developer with expertise in React, JavaScript, and UI/UX design.</p>
      <h3>Skills</h3>
      <ul>
        <li>React.js</li>
        <li>JavaScript (ES6+)</li>
        <li>HTML5 & CSS3</li>
        <li>Node.js & Express</li>
      </ul>
    </section>
  );
}

export default About;
```

- **Lists key skills dynamically** using <ul> lists.

4.3 Creating the Experience Section

File: src/components/Experience.js

```
import React from "react";
import styles from "./Experience.module.css";

function Experience() {
  return (
    <section id="experience" className={styles.experience}>
      <h2>Work Experience</h2>
      <div>
        <h3>Frontend Developer - ABC Company</h3>
        <p>Jan 2021 - Present</p>
        <ul>
          <li>Developed and maintained React applications.</li>
          <li>Optimized UI performance by 30%.</li>
        </ul>
      </div>
    </section>
  );
}

export default Experience;
```

- **Showcases work history** in an organized format.

4.4 Creating the Projects Section

File: src/components/Projects.js

```
import React from "react";
import styles from "./Projects.module.css";

function Projects() {
  const projects = [
    { name: "Portfolio Website", link: "https://myportfolio.com" },
    { name: "E-commerce App", link: "https://shopapp.com" },
  ];

  return (
    <section id="projects" className={styles.projects}>
      <h2>Projects</h2>
      <ul>
        {projects.map((project, index) => (
          <li key={index}>
            <a href={project.link} target="_blank" rel="noopener
noreferrer">
```

```
                {project.name}
              </a>
            </li>
          ))}
        </ul>
      </section>
    );
}

export default Projects;
```

- **Dynamically renders projects** from an array.

4.5 Creating the Contact Section

File: src/components/Contact.js

```
import React from "react";
import styles from "./Contact.module.css";

function Contact() {
  return (
    <section id="contact" className={styles.contact}>
      <h2>Contact Me</h2>
```

```
        <p>Email: johndoe@example.com</p>
        <p>LinkedIn: <a
href="https://linkedin.com/in/johndoe">linkedin.com/in/johndoe</a
></p>
      </section>
    );
}

export default Contact;
```

- **Displays contact details** with social links.

CHAPTER 5: STYLING THE RESUME WEBSITE WITH CSS MODULES

5.1 Example: Styling the Header (Header.module.css)

```
.header {
  display: flex;
  justify-content: space-between;
  background: #282c34;
  padding: 20px;
  color: white;
}

.header nav a {
```

```
  color: white;
  margin: 0 10px;
  text-decoration: none;
}
```

- **Keeps styles scoped** to prevent global CSS conflicts.

Chapter 6: Deploying the Resume Website Online

6.1 Deploying with Netlify

1. Run npm run build to create an optimized production build.
2. Sign up on https://www.netlify.com/.
3. Drag and drop the build folder to Netlify.

6.2 Deploying with Vercel

1. Install Vercel CLI:
2. npm install -g vercel
3. Run deployment command:
4. vercel

Chapter 7: Exercises for Hands-on Practice

1. **Enhance the Resume Page** by adding an **education section**.
2. **Implement Dark Mode** using React useState.
3. **Improve UI with Animations** using **CSS transitions**.

CONCLUSION

By building a **React-based Resume Website**, you have practiced:

✅ **React components and JSX**
✅ **CSS Modules for styling**
✅ **React Router for navigation**
✅ **Deploying React projects online**

🚀 **Next Steps:**

- Add **form validation** for the contact section.
- Optimize for **mobile responsiveness** using media queries.
- Implement a **backend to store user messages**!

HANDS-ON PRACTICE: BUILD A TO-DO APP WITH USESTATE

In this hands-on project, we will build a **To-Do App** using React's useState hook. The app will allow users to **add, delete, and mark tasks as completed** dynamically.

PROJECT FEATURES

✅ **Add tasks** to the list.
✅ **Mark tasks as completed**.
✅ **Delete tasks from the list**.
✅ **Persist tasks using local storage (optional for advanced users).**

Step 1: Setup the React App

If you don't have a React project set up, create one using:

npx create-react-app todo-app

cd todo-app

npm start

Inside the src/ folder, create a new file called **ToDoApp.js**.

Step 2: Create the To-Do App Component

ToDoApp.js – Full Implementation

import React, { useState } from "react";

```
import styled from "styled-components";


// Styled components for UI
const Container = styled.div`
  width: 400px;
  margin: 50px auto;
  padding: 20px;
  border-radius: 10px;
  background: #f8f9fa;
  box-shadow: 0px 0px 10px rgba(0, 0, 0, 0.1);
`;


const Input = styled.input`
  width: 70%;
  padding: 10px;
  margin-right: 10px;
  border: 1px solid #ccc;
  border-radius: 5px;
`;


const Button = styled.button`
```

```
  padding: 10px 15px;
  border: none;
  background-color: #28a745;
  color: white;
  font-weight: bold;
  border-radius: 5px;
  cursor: pointer;
  &:hover {
    background-color: #218838;
  }
`;

const TaskList = styled.ul`
  list-style: none;
  padding: 0;
`;

const TaskItem = styled.li`
  display: flex;
  justify-content: space-between;
  align-items: center;
```

```
  padding: 10px;
  margin: 10px 0;
  border-radius: 5px;
  background-color: ${(props) => (props.completed ? "#d4edda" : "white")};
  border: ${(props) => (props.completed ? "2px solid #28a745" : "2px solid #ccc")};
`;

const DeleteButton = styled.button`
  background: #dc3545;
  color: white;
  border: none;
  padding: 5px 10px;
  border-radius: 5px;
  cursor: pointer;
  &:hover {
    background: #c82333;
  }
`;

// Main ToDoApp Component
```

```
function ToDoApp() {
  const [tasks, setTasks] = useState([]);
  const [newTask, setNewTask] = useState("");

  // Function to add a task
  const addTask = () => {
    if (newTask.trim() === "") return;
    setTasks([...tasks, { text: newTask, completed: false }]);
    setNewTask("");
  };

  // Function to toggle completion status
  const toggleCompletion = (index) => {
    const updatedTasks = [...tasks];
    updatedTasks[index].completed =
!updatedTasks[index].completed;
    setTasks(updatedTasks);
  };

  // Function to delete a task
  const deleteTask = (index) => {
```

```
    const updatedTasks = tasks.filter((_, i) => i !== index);
    setTasks(updatedTasks);
  };

  return (
    <Container>
      <h2>To-Do List</h2>
      <div>
        <Input
          type="text"
          placeholder="Add a new task..."
          value={newTask}
          onChange={(e) => setNewTask(e.target.value)}
        />
        <Button onClick={addTask}>Add</Button>
      </div>
      <TaskList>
        {tasks.map((task, index) => (
          <TaskItem key={index} completed={task.completed}>
            <span
              onClick={() => toggleCompletion(index)}
```

```
                    style={{ cursor: "pointer", textDecoration:
task.completed ? "line-through" : "none" }}
                  >
                    {task.text}
                  </span>
                  <DeleteButton onClick={() =>
deleteTask(index)}>Delete</DeleteButton>
                </TaskItem>
              ))}
            </TaskList>
          </Container>
    );
}

export default ToDoApp;
```

Step 3: Integrate ToDoApp Component in App.js

Replace the contents of src/App.js with:

```
import React from "react";
import ToDoApp from "./ToDoApp";

function App() {
```

```
  return (
    <div>
      <ToDoApp />
    </div>
  );
}


export default App;
```

Step 4: Run the Application

Start the React app:

```
npm start
```

Go to http://localhost:3000 in your browser. 🎉 Your **To-Do List App** is now fully functional!

Step 5: Bonus - Persisting Tasks in Local Storage

To keep tasks saved even after refreshing the page, modify ToDoApp.js to include useEffect for local storage.

Updated Code (with Local Storage)

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

import styled from "styled-components";
```

```
// Styled components for UI
const Container = styled.div`
  width: 400px;
  margin: 50px auto;
  padding: 20px;
  border-radius: 10px;
  background: #f8f9fa;
  box-shadow: 0px 0px 10px rgba(0, 0, 0, 0.1);
`;

// Remaining styles here (same as previous code)...

function ToDoApp() {
  const [tasks, setTasks] = useState([]);
  const [newTask, setNewTask] = useState("");

  // Load tasks from local storage when app starts
  useEffect(() => {
    const storedTasks = JSON.parse(localStorage.getItem("tasks"));
    if (storedTasks) setTasks(storedTasks);
  }, []);
```

```
  // Save tasks to local storage whenever they change
  useEffect(() => {
    localStorage.setItem("tasks", JSON.stringify(tasks));
  }, [tasks]);

  const addTask = () => {
    if (newTask.trim() === "") return;
    setTasks([...tasks, { text: newTask, completed: false }]);
    setNewTask("");
  };

  const toggleCompletion = (index) => {
    const updatedTasks = [...tasks];
    updatedTasks[index].completed =
!updatedTasks[index].completed;
    setTasks(updatedTasks);
  };

  const deleteTask = (index) => {
    const updatedTasks = tasks.filter((_, i) => i !== index);
```

```
    setTasks(updatedTasks);
  };


  return (
    <Container>
      <h2>To-Do List</h2>
      <div>
        <input type="text" placeholder="Add a new task..."
value={newTask} onChange={(e) => setNewTask(e.target.value)} />
        <button onClick={addTask}>Add</button>
      </div>
      <ul>
        {tasks.map((task, index) => (
          <li key={index} style={{ textDecoration: task.completed ?
"line-through" : "none" }}>
            <span onClick={() =>
toggleCompletion(index)}>{task.text}</span>
            <button onClick={() =>
deleteTask(index)}>Delete</button>
          </li>
        ))}
      </ul>
    </Container>
```

```
  );
}

export default ToDoApp;
```

✅ **Tasks are now saved even after page reload!**

Final Thoughts

🎯 What You Learned

✓ How to use useState for managing **stateful data**.
✓ Handling user input with onChange.
✓ Creating dynamic UI with **styled-components**.
✓ Using useEffect for **local storage persistence**.

🚀 Next Steps

- Add a **due date feature** to each task.
- Implement a **task filtering system** (All, Completed, Pending).
- Create a **dark mode toggle** using **styled-components themes**.

By completing this **hands-on project,** you have taken a big step in mastering **React Hooks** and **state management**. Keep building! 🚀🔥

ASSIGNMENTS:

- DEVELOP A FULLY RESPONSIVE TO-DO LIST USING REACT
- IMPLEMENT NAVIGATION USING REACT ROUTER

Assignment Solution: Develop a Fully Responsive To-Do List Using React

Objective

The goal of this assignment is to build a **fully functional and responsive To-Do List application** using **React.js** and **useState Hook**. The app should allow users to:

- ✅ Add new tasks
- ✅ Mark tasks as completed
- ✅ Delete tasks
- ✅ Persist tasks using Local Storage
- ✅ Ensure a responsive design with Tailwind CSS or CSS Modules

---

Project Setup

Step 1: Create a React App

If you don't have a React project, create one using:

```
npx create-react-app todo-app
cd todo-app
npm start
```

Step 2: Install Dependencies

To enhance styling and responsiveness, install Tailwind CSS (optional):

```
npm install tailwindcss
npx tailwindcss init -p
```

Configure tailwind.config.js:

```
module.exports = {
 content: ["./src/**/*.{js,jsx,ts,tsx}"],
 theme: { extend: {} },
 plugins: [],
};
```

Add Tailwind to index.css:

```
@tailwind base;
```

```
@tailwind components;
@tailwind utilities;
```

Step 3: Building the To-Do List App

1. Create a ToDo.js Component

Create a new file: src/components/ToDo.js

ToDo.js (Main Component)

```
import React, { useState, useEffect } from "react";
import "./ToDo.css"; // Optional CSS for custom styling

function ToDo() {
  const [tasks, setTasks] = useState([]);
  const [newTask, setNewTask] = useState("");

  // Load tasks from local storage on component mount
  useEffect(() => {
    const storedTasks = JSON.parse(localStorage.getItem("tasks"));
    if (storedTasks) setTasks(storedTasks);
  }, []);

  // Save tasks to local storage whenever tasks change
  useEffect(() => {
    localStorage.setItem("tasks", JSON.stringify(tasks));
  }, [tasks]);

  // Function to add a task
  const addTask = () => {
    if (newTask.trim() === "") return;
    setTasks([...tasks, { id: Date.now(), text: newTask, completed: false }]);
```

```
    setNewTask("");
  };

  // Function to toggle task completion
  const toggleTask = (id) => {
    setTasks(
      tasks.map(task =>
        task.id === id ? { ...task, completed: !task.completed } : task
      )
    );
  };

  // Function to delete a task
  const deleteTask = (id) => {
    setTasks(tasks.filter(task => task.id !== id));
  };

  return (
    <div className="todo-container">
      <h1>To-Do List</h1>
      <div className="input-section">
        <input
          type="text"
          value={newTask}
          onChange={(e) => setNewTask(e.target.value)}
          placeholder="Enter a task..."
        />
        <button onClick={addTask}>Add Task</button>
      </div>
```

```
        <ul className="task-list">
          {tasks.map(task => (
            <li key={task.id} className={task.completed ? "completed" : ""}>
              <span onClick={() => toggleTask(task.id)}>{task.text}</span>
              <button onClick={() => deleteTask(task.id)}>❌</button>
            </li>
          ))}
        </ul>
      </div>
    );
}

export default ToDo;
```

---

2 Create ToDo.css for Styling

Create a ToDo.css file in src/components/ and add the following styles:

```
.todo-container {
  max-width: 400px;
  margin: auto;
  text-align: center;
  background: #f8f9fa;
  padding: 20px;
  border-radius: 10px;
  box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
}

.input-section {
  display: flex;
  gap: 10px;
```

```
  margin-bottom: 20px;
}

.input-section input {
  flex: 1;
  padding: 10px;
  border: 2px solid #ddd;
  border-radius: 5px;
}

.input-section button {
  background: #007bff;
  color: white;
  border: none;
  padding: 10px 15px;
  border-radius: 5px;
  cursor: pointer;
}

.task-list {
  list-style: none;
  padding: 0;
}

.task-list li {
  display: flex;
  justify-content: space-between;
  padding: 10px;
  background: white;
```

```
  margin: 5px 0;
  border-radius: 5px;
  box-shadow: 0 2px 5px rgba(0, 0, 0, 0.1);
}

.task-list li.completed span {
  text-decoration: line-through;
  color: gray;
}

.task-list button {
  background: none;
  border: none;
  cursor: pointer;
  font-size: 18px;
}
```

---

3. Use the To-Do Component in App.js

Modify src/App.js to use the **ToDo** component.

```
import React from "react";
import ToDo from "./components/ToDo";

function App() {
  return (
    <div className="app-container">
      <ToDo />
    </div>
  );
}
```

```
export default App;
```

Step 4: Testing & Running the Application

Run your React app to test the functionality:

```
npm start
```

🚀 **Your To-Do App is now fully functional!** 🚀

Enhancements & Bonus Challenges

1Add a Dark Mode Feature

- Use useState to toggle dark mode.
- Modify styles dynamically with conditional classes.

2 Implement Filter Options

- Add "All", "Completed", and "Pending" filters.

3 Drag & Drop Task Reordering

- Use react-beautiful-dnd for **drag-and-drop functionality**.

4 Use Local Storage for Persistent Data

- Ensure tasks are **saved and retrieved** when the user refreshes the page.

Assignment Submission Guidelines

✅ Submit your GitHub repository link containing the **React To-Do App code**.
✅ Deploy your app on **Vercel or Netlify** and provide the live URL.
✅ Write a **README.md** file with setup instructions.

Conclusion

By completing this assignment, you have successfully built a **functional and responsive To-Do List App** using React. You have practiced using:

✅ useState for state management
✅ useEffect for persistent data storage

✅ **CSS Modules or Tailwind for styling**
✅ **Dynamic UI updates and event handling**

🚀 **Next Steps:**

- Add **React Router** for a multi-page app.
- Integrate **Firebase or a database** for cloud storage.
- Implement **authentication (login/logout)**.

🔹 **Once completed, share your project in your portfolio!** 🎉

Assignment Solution: Implement Navigation Using React Router

Objective

The goal of this assignment is to **implement navigation using React Router** to create a **multi-page React application**. You will set up a **React Router navigation system** that allows users to switch between different sections of a **resume website** without a full-page reload.

Instructions

Step 1: Install React Router

If you haven't installed React Router yet, run the following command:

npm install react-router-dom

Step 2: Create the Required Components

Inside the src/components/ folder, create the following components:

1. **Home.js** – Displays an introduction.
2. **About.js** – Contains profile and skills information.
3. **Experience.js** – Lists work experience.
4. **Projects.js** – Displays completed projects.
5. **Contact.js** – Shows contact details.
6. **Navbar.js** – Provides navigation links.

Step 3: Set Up React Router in App.js

Modify src/App.js to include React Router:

import React from "react";

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

import Navbar from "./components/Navbar";

```
import Home from "./components/Home";
import About from "./components/About";
import Experience from "./components/Experience";
import Projects from "./components/Projects";
import Contact from "./components/Contact";

function App() {
  return (
    <Router>
      <Navbar />
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
        <Route path="/experience" element={<Experience />} />
        <Route path="/projects" element={<Projects />} />
        <Route path="/contact" element={<Contact />} />
      </Routes>
    </Router>
  );
}

export default App;
```

💡 **Key Takeaways**:
✅ Router – Wraps the entire app to enable routing.
✅ Routes – Groups multiple Route components.
✅ Route – Defines different paths and components to render.

Step 4: Create the Navigation Bar (Navbar.js)

File: src/components/Navbar.js

```
import React from "react";
```

```
import { Link } from "react-router-dom";
import styles from "./Navbar.module.css";  // Using CSS Modules

function Navbar() {
  return (
    <nav className={styles.navbar}>
      <ul>
        <li><Link to="/">Home</Link></li>
        <li><Link to="/about">About</Link></li>
        <li><Link to="/experience">Experience</Link></li>
        <li><Link to="/projects">Projects</Link></li>
        <li><Link to="/contact">Contact</Link></li>
      </ul>
    </nav>
  );
}

export default Navbar;
```

💡 **Why use <Link> instead of <a>?**

- Prevents **full-page reloads**.
- Maintains **single-page application (SPA) behavior**.

Step 5: Define Each Page Component

Home.js

```
import React from "react";

function Home() {
  return (
    <section>
```

```
        <h1>Welcome to My Resume Website</h1>
        <p>This site showcases my skills, experience, and projects.</p>
      </section>
    );
}

export default Home;
```

About.js

```
import React from "react";

function About() {
  return (
    <section>
      <h2>About Me</h2>
      <p>I'm a passionate frontend developer with experience in React.js, JavaScript, and UI/UX design.</p>
    </section>
  );
}

export default About;
```

Experience.js

```
import React from "react";

function Experience() {
  return (
    <section>
      <h2>Work Experience</h2>
      <p>Frontend Developer at XYZ Company - 2021 to Present</p>
    </section>
```

```
  );
}

export default Experience;
```

Projects.js

```
import React from "react";

function Projects() {
  return (
    <section>
      <h2>Projects</h2>
      <ul>
        <li>Portfolio Website</li>
        <li>E-commerce Web App</li>
      </ul>
    </section>
  );
}

export default Projects;
```

Contact.js

```
import React from "react";

function Contact() {
  return (
    <section>
      <h2>Contact Me</h2>
      <p>Email: johndoe@example.com</p>
      <p>LinkedIn: linkedin.com/in/johndoe</p>
```

```
      </section>
    );
}

export default Contact;
```

Step 6: Style the Navigation Bar Using CSS Modules

File: src/components/Navbar.module.css

```
.navbar {
    background: #333;
    padding: 15px;
}

.navbar ul {
    list-style: none;
    display: flex;
    justify-content: center;
}

.navbar li {
    margin: 0 15px;
}

.navbar a {
    color: white;
    text-decoration: none;
    font-size: 18px;
}
```

```
.navbar a:hover {
  color: lightgray;
}
```

Bonus Challenge: Add an Active Navigation Link Style

React Router provides a NavLink component that allows **active styling** when a link is selected.

Modify Navbar.js like this:

```
import React from "react";
import { NavLink } from "react-router-dom";
import styles from "./Navbar.module.css";

function Navbar() {
  return (
    <nav className={styles.navbar}>
      <ul>
        <li><NavLink to="/" className={({ isActive }) => isActive ? styles.active : ""}>Home</NavLink></li>
        <li><NavLink to="/about" className={({ isActive }) => isActive ? styles.active : ""}>About</NavLink></li>
        <li><NavLink to="/experience" className={({ isActive }) => isActive ? styles.active : ""}>Experience</NavLink></li>
        <li><NavLink to="/projects" className={({ isActive }) => isActive ? styles.active : ""}>Projects</NavLink></li>
        <li><NavLink to="/contact" className={({ isActive }) => isActive ? styles.active : ""}>Contact</NavLink></li>
      </ul>
    </nav>
  );
}
```

```
export default Navbar;
```

Modify Navbar.module.css to include active link styles:

```
.active {
  font-weight: bold;
  border-bottom: 2px solid white;
}
```

✅ **Active link styling is now applied dynamically!**

Step 7: Test Your Application

Run the development server and test the navigation:

```
npm start
```

1. Click each navigation link and check if the correct page loads.
2. Ensure that no **full-page reloads** happen when navigating.
3. Verify the active link styling.

Assignment Submission

- **Submit the GitHub repository link** containing your project.
- **Deploy the project** using **Netlify or Vercel** and submit the deployed link.
- **Take a short video recording** (optional) demonstrating navigation functionality.

Conclusion

You have successfully implemented **React Router for navigation** in a resume website! This assignment helped you practice:

✅ **Setting up routing in React**
✅ **Creating multiple page components**
✅ **Navigating using <Link> and <NavLink>**
✅ **Styling the navigation bar with CSS Modules**

🚀 **Next Steps:**

- Add **nested routes** for projects or experience details.
- Implement **protected routes** for a login system.

- Add **dynamic routes** to display detailed user profiles!

🔥 **Ready for more challenges?** Try implementing a **404 error page** when users navigate to an unknown route! 🚀