

Full Stack Development BootCamp - Day 5

Build Real-World Apps from Backend to Frontend!

Trainer:

Muhammad Maaz Sheikh

Technical Team Lead



Tech Stack Covered:

- Nest Js
- PostgreSQL
- React Js
- Real world project development

Organized By:

Iqra University

Recap Day 1,2,3,4





Table of Content

Section A-6: Handling Cross-Origin Issues (Nest JS)

- What is CORS?
- Enable CORS globally
- Custom CORS Configuration
- Environment based Configuration

Section A-7: Introduction to React

- What is React, History and Evolution
- Why use React?
- Setup & Installation
- Creating a new React project. Vite
- Understanding the React Project Structure
- Core Concepts (Components, Props, State, Lifecycle)
- React Event Handling
- React Conditional Rendering
- Hooks (useState, useEffect, useContext)
- Creating form in React



Table of Content

Section A-8: Routing With React Router

- Setting up React Router (v6+)
- Navigating between pages (Link, NavLink, useNavigate)
- Route Parameters (Dynamic Routing) and Query Strings
- Protected Routes and Role-Based Access
- Nested Routes
- Handling 404 (Not Found) Pages.

Handling Cross-Origin Resource Sharing (CORS) Issues.



Handling Cross-Origin Resource Sharing (CORS)

What is CORS?

CORS is a browser security feature that restricts web pages from making requests to a different domain than the one that served the web page.

If your frontend and backend are on different ports or domains (e.g., React on **localhost:3000** and NestJS on **localhost:3001**), you'll need to enable CORS.

✓ Option 1: Enable CORS Globally (Basic Setup)

Update your **main.ts** file:

This enables CORS for **all origins** by default.

```
// main.ts
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);

  app.enableCors(); // <--- Simple CORS enable

  await app.listen(3001);
}
bootstrap();
```



Handling Cross-Origin Resource Sharing (CORS)

✓ Option 2: Custom CORS Configuration

```
app.enableCors({  
  origin: 'http://localhost:3000', // Frontend URL  
  methods: 'GET,POST,PUT,DELETE',  
  credentials: true, // if you're using cookies or Authorization headers  
});
```

✓ Option 3: Environment-based Setup

```
app.enableCors({  
  origin: process.env.FRONTEND_URL || '*',  
});
```

Handling Cross-Origin Resource Sharing (CORS)

Example Use Case


- Frontend (React app at `http://localhost:3000`)
- Backend (NestJS at `http://localhost:3001`)

main.ts

```
app.enableCors({  
  origin: 'http://localhost:3000',  
  credentials: true,  
});
```

This setup ensures that requests from your frontend app can be processed by your Nest JS API.

Troubleshooting

- **Error:** Access to fetch at '`http://localhost:3001/api`' from origin '`http://localhost:3000`' has been blocked by CORS policy
 -  **Solution:** Set **origin** correctly in **enableCors()**
- If using cookies or tokens: make sure **credentials: true** is set, and the frontend sets credentials: 'include' in fetch/axios.

Introduction to React





Introduction to React

What is React?

React is an open source **JavaScript** library used to build **user interfaces (UIs)** — especially for **single-page applications (SPAs)** where you need a dynamic and fast user experience. It updated the content of the page dynamically without reloading the page.

Key Points:

- Developed and maintained by **Facebook (now Meta)**
- Focuses on the **View Layer** in the **MVC** architecture
- Uses a **component-based** architecture: build reusable UI pieces
- Powered by a **Virtual DOM** for fast rendering



Introduction to React

Why Use React?

Component-based Architecture:

- React JS follows a component-based architecture, where UIs are divided into reusable components.
- Traditional web development often involves a mix of Html, Css and JavaScript, leading to tightly coupled code.
- React's component-based approach promotes modularity, reusability, and easier maintenance of code.

Unidirectional Data Flow:

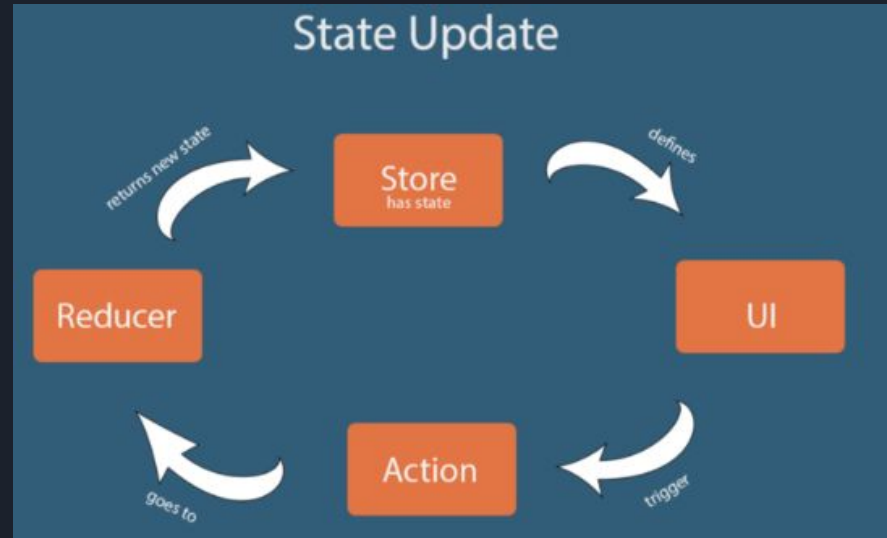
- React JS enforces a unidirectional data flow, also known as one-way binding.
- Traditional web development often involves two-way data binding, where changes in one part of the application affect others, making it harder to track and manage data changes.
- React's unidirectional data flow simplifies data management, reducing the likelihood of bugs and making the application easier to reason about.

Introduction to React

Why Use React?

State Management:

- State management is a critical aspect of building complex web applications.
- React Js provides several options for managing state efficiently within its ecosystem.
- Various ways of State Management:
 - Local Component State
 - use state
 - Context API
 - Redux



Setup & Installation





Setup & Installation

Step 1: Install Node.js and npm

For start creating app we first need the Node installed on our system. You can check if node is installed and which version of node you have

- Download **node.js** from: <https://nodejs.org/>
- **Verify installation:** After installation, open a terminal and run:

```
node -v  
npm -v
```

- You should see the version numbers of **Node.js** and **npm**, confirming the installation was successful.

Step 2: Create a React App Using Vite

Vite is a fast and modern build tool that makes React development blazing fast.

- Open your terminal and run the following command:

```
npm create vite@latest
```

- Replace **react-fundamentals** with your desired project name i.e. **my-react-app**.
- Select the framework and variant when prompted: **Select React and typescript**



Setup & Installation

Step 3: Navigate to Project folder and run following commands

- `cd` into the vite project (my-react-app)
- `npm install`
- `npm run dev`

```
C:\FullStackDevelopment\ReactJs>npm create vite@latest
|
o Project name:
| my-react-app
|
o Select a framework:
| React
|
o Select a variant:
| TypeScript
|
o Scaffolding project in C:\FullStackDevelopment\ReactJs\my-react-app...
|
- Done. Now run:

cd my-react-app
npm install
npm run dev
```



Understanding React Project Structure

Source Code Files:

File	Description
<code>main.tsx</code>	Entry point of the React App. Renders <code><App /></code> to the Dom.
<code>app.tsx</code>	Root component where your app starts.
<code>app.css/index.css</code>	App styling files
<code>assets/</code>	Place to store images, logos, etc.

Other Important Files:

File	Purpose
<code>index.html</code>	Root HTML file, very minimal — Vite injects your React app into it
<code>package.json</code>	Lists dependencies and scripts like <code>npm run dev</code>
<code>vite.config.js</code>	Vite's configuration file (optional unless you need custom config)
<code>.gitignore</code>	Tells Git which files/folders to ignore

Core Concepts (Components, Props, State, Lifecycle)

1. Components

✓ What is a Component?

A component is a reusable, independent piece of UI in React. Think of it like a function that returns HTML (JSX).

Step 1: Create a New File for the Component

In your project inside the **src** folder:

```
src/  
├── components/  
│   └── Welcome.tsx
```

Step 2: Create a Functional Component

Create a function in welcome.tsx file

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

🧠 What's Happening?

- **Welcome** is a React component
- You define a function that returns JSX (HTML in JS).
- `export default` makes it usable in other files.
- Can be reused anywhere in your app.



Core Concepts (Components, Props, State, Lifecycle)

1. Components

Step 3: Use the Component in App.tsx

Create a function in welcome.tsx file

```
import Welcome from './components/Welcome';
```

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

```
export default App;
```

🧠 What's Happening?

- `welcome` is imported and used like an HTML tag: `<Welcome />`



Core Concepts (Components, Props, State, Lifecycle)

2. Props

✓ What are Props?

Props are like function parameters. They allow you to pass data from parent to child component.

Step 1: Define Props in your already created Component

Open your welcome component:

```
function Welcome(props) {  
  return <h1>Hello, {props.name}!</h1>;  
}  
  
export default Welcome;
```

You're seeing an error on props because you're using TypeScript, and the props need to be typed properly. In TypeScript, you must define the shape of the props the component will receive.

✓ Fixed TypeScript Version

```
type WelcomeProps = {  
  name: string;  
};  
  
function Welcome(props: WelcomeProps) {  
  return <h1>Hello, {props.name}!</h1>;  
}  
  
export default Welcome;
```

Core Concepts (Components, Props, State, Lifecycle)

2. Props

3.

Step 2: Passing Props when Calling Welcome from App.tsx:

Create a function in **welcome.tsx** file

```
function App() {  
  return (  
    <div>  
      <Welcome name="Maaz" />  
    </div>  
  );  
}
```



Explanation:

- **name="Maaz"** is a **prop**
- **Welcome** receives it as **props.name**
- **Output: Hello, Maaz!**

Core Concepts (Components, Props, State, Lifecycle)

3. State

✓ What is State?

State is like a variable that belongs to a component. It stores dynamic data and controls behavior/UI updates.

Step 1: Create a New File for the Component

Inside the **src/components** folder, create a new file: **Counter.tsx**

```
src/  
├── components/  
│   └── Counter.tsx
```

💡 You can create the file manually or via terminal:

System	Create file command
Linux/macOS	<code>touch src/components/UserForm.tsx</code>
Windows PowerShell	<code>ni src\components\UserForm.tsx</code>
Windows CMD	<code>echo.> src\components\UserForm.tsx</code>

Core Concepts (Components, Props, State, Lifecycle)

3. State

Step 2: Example with `useState`:

Create a function in `counter.tsx` file

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

function Counter() {
  const [count, setCount] = useState(0); // Initial state

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

export default Counter;
```



Explanation:

- `useState(0)` initializes `count` to 0
- `setCount` is a function to update `count`
- `setCount(count + 1)` Trigger/updates state when button is clicked
- `{count}` Display the current state in JSX
- Clicking the button updates the UI

Core Concepts (Components, Props, State, Lifecycle)

3. State

Step 3: Example with `useState`:

You can check the reflection and output of the counter.tsx file by using this in app.tsx

Your App.tsx should look like this.

```
import { useState } from 'react'
import './App.css'
import Welcome from './components/welcome'
import Counter from './components/counter'

function App() {
  return (
    <>
      <div>
        <Welcome name="Maaz"></Welcome>
        <Counter/>
      </div>
    </>
  )
}

export default App
```




Core Concepts (Components, Props, State, Lifecycle)

4. Lifecycle (Functional & Class Component)

In functional components, lifecycle logic is handled using Hooks like `useEffect`

✓ What is Lifecycle?

Lifecycle refers to the sequence of stages a React component goes through:

Phase	Description
 Mounting	Component is created and inserted into the DOM
 Updating	State or props change and component re-renders
 Unmounting	Component is removed from the DOM

Step 1: Create a New File

Inside the `src/components` folder, create a new file: **LifeCycleExample.tsx**

Core Concepts (Components, Props, State, Lifecycle)

4. Lifecycle (Functional Component)

Step 2: Add Code with `useEffect`

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

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

  // Component Did Mount + Update
  useEffect(() => {
    console.log('🟢 Component Mounted or Updated');

    // Component Will Unmount
    return () => {
      console.log('🔴 Cleanup before next update or unmount');
    };
  }, [count]); // Runs when `count` changes

  return (
    <div>
      <h2>Lifecycle Demo</h2>
      <p>Click Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increase</button>
    </div>
  );
}

export default LifecycleExample
```

Core Concepts (Components, Props, State, Lifecycle)

4. Lifecycle (Functional Component)

Step 3: Use it in App.tsx

```
// src/App.jsx
import LifecycleExample from './components/LifecycleExample';

function App() {
  return (
    <div>
      <h1>React Lifecycle (Functional)</h1>
      <LifecycleExample />
    </div>
  );
}

export default App;
```

What's Happening?

Hook	Acts like	Purpose
useEffect()	componentDidMount + Update	Run code after render/update
Return () =>	componentWillUnmount	Cleanup before re-render or unmount
[count]	Dependency Array	Run only when count changes

Core Concepts (Components, Props, State, Lifecycle)

4. Lifecycle (Class Component)

Step 1: Add Code in **Class Component**

Inside the **src/components** folder, create a new file: **LegacyLifecycleExample.tsx**

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

class LegacyLifecycle extends Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
    console.log("🔗 Constructor");
  }

  componentDidMount() {
    console.log("Component Mounted");
  }

  componentDidUpdate() {
    console.log("Component Updated");
  }

  componentWillUnmount() {
    console.log("Component Will Unmount");
  }

  render() {
    return (
      <div>
        <h2>Legacy Lifecycle Demo</h2>
        <p>Click Count: {this.state.count}</p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Increase
        </button>
      </div>
    );
  }
}

export default LegacyLifecycle;
```



Core Concepts (Components, Props, State, Lifecycle)

Summary: Lifecycle Methods

Phase	Functional Component (<code>useEffect</code>)	Class Component
Mount	<code>useEffect(() => {...}, [])</code>	<code>componentDidMount()</code>
Update	<code>useEffect(() => {...}, [dep])</code>	<code>componentDidUpdate()</code>
Unmount	<code>return () => {...} inside useEffect</code>	<code>componentWillUnmount()</code>



React Event Handling

What are Events?

In web development, an event refers to anything that happens on a web page, such as when a button is clicked or a form is submitted. In React, you can handle these events with event handlers.

Why is Event Handling important?

Event handling is crucial in creating dynamic web applications. By responding to events with custom logic, you can create user interactions that make your application feel more responsive and intuitive.

React Event Handling

React components can listen and respond to various events similar to HTML event handling. Events are handled using event handlers, which are functions that are executed when a specific event occurs.

We have some examples in React demonstrating how to listen and respond to various events. Below are some Event Handling types in React.

1. Handling Click Events
2. Handling Input Events (onChange)
3. Handling Form Submission
4. Handling Mouse Events
5. Handling Keyboard Events
6. Handling Focus Events



React Event Handling

1. Handling Click Events:

In this example, when the button is clicked, the handleClick method is invoked, which displays an alert.

Steps:

Create new file with name **ClickEventExample.tsx** in **src/components/events** and add below line of code in file.

```
import React from "react";

class ClickEventExample extends React.Component {
  handleClick() {
    alert("Button clicked!");
  }
  render() {
    return <button onClick={this.handleClick}>Click me</button>;
  }
}

export default ClickEventExample;
```



React Event Handling

2. Handling Input Events (onChange):

In this example, the `handleChange` method is called whenever the input field's value changes. It updates the component's state with the new value entered by the user

Steps:

Create new file with name **NameInputEventExample.tsx** and add below line of code in file.

Code:

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

function NameInputEventExample() {
  const [name, setName] = useState<string>('');

  const handleChange = (e: React.ChangeEvent<HTMLInputElement>): void => {
    setName(e.target.value);
  };

  return (
    <div>
      <label>Your Name: </label>
      <input type="text" value={name} onChange={handleChange} />
      <p>Hello, {name}</p>
    </div>
  );
}

export default NameInputEventExample;
```

Type: `React.ChangeEvent<HTMLInputElement>`

Why: Safely accesses `e.target.value`.



React Event Handling

3. Handling Form Submission:

Use Case: Simple Contact form.

Steps:

Create new file with name **contactFormEventExample.tsx** and add below line of code in file.

Code:

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

function ContactFormEventExample() {
  const [email, setEmail] = useState('');

  const handleSubmit = (e: React.FormEvent<HTMLFormElement>): void => {
    e.preventDefault(); // Prevent reload
    alert(`Submitted: ${email}`);
    setEmail('');
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="email"
        value={email}
        onChange={(e) => setEmail(e.target.value)}
        placeholder="Enter email"
      />
      <button type="submit">Submit</button>
    </form>
  );
}

export default ContactFormEventExample;
```

Type: React.FormEvent<HTMLFormElement>

Why: Prevents reload, handles form logic.



React Event Handling

4. Handling Mouse Events:

Use Case: Show coordinates on hover.

Steps:

Create new file with name **MouseTrackerEventExample.tsx** and add below line of code in file.

Code:

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

function MouseTrackerEventExample() {
  const [coords, setCoords] = useState({ x: 0, y: 0 });

  const handleMouseMove = (e: React.MouseEvent<HTMLDivElement>): void => {
    setCoords({ x: e.clientX, y: e.clientY });
  };

  return (
    <div onMouseMove={handleMouseMove} style={{ height: '200px', background: '#eef' }}>
      <p>Mouse Position: {coords.x}, {coords.y}</p>
    </div>
  );
}

export default MouseTrackerEventExample;
```

Type: React.MouseEvent<HTMLDivElement>

Why: Tracks user's mouse within a component.



React Event Handling

5. Handling Keyboard Events:

Use Case: Detect Enter key in input field.

Steps:

Create new file with name **KeyPressInputEventExample.tsx** and add below line of code in file.

Code:

```
function KeyPressInputEventExample() {  
  const handleKeyDown = (e: React.KeyboardEvent<HTMLInputElement>): void => {  
    if (e.key === 'Enter') {  
      alert('Enter pressed!');  
    }  
  };  
  
  return <input type="text" onKeyDown={handleKeyDown} placeholder="Type and press Enter" />;  
}  
  
export default KeyPressInputEventExample
```

Type: React.KeyboardEvent<HTMLInputElement>

Why: Detects and responds to keyboard interactions.



React Event Handling

6. Handling Focus Events:

Use Case: Show helper text when input is focused.

Steps:

Create new file with name **FocusInputEventExample.tsx** and add below line of code in file.

Code:

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

function FocusInputEventExample () {
  const [focused, setFocused] = useState<boolean>(false);

  const handleFocus = (e: React.FocusEvent<HTMLInputElement>): void => setFocused(true);
  const handleBlur = (e: React.FocusEvent<HTMLInputElement>): void => setFocused(false);

  return (
    <div>
      <input type="text" onFocus={handleFocus} onBlur={handleBlur} />
      {focused && <p style={{ color: 'blue' }}>Start typing your name... </p>}
    </div>
  );
}

export default FocusInputEventExample ;
```

Type: React.FocusEvent<HTMLInputElement>

Why: Useful for validation hints and UI changes.



React Conditional Rendering

What is Condition Rendering?

Conditional rendering allows dynamic control over which UI elements or content are displayed based on specific conditions. It is commonly used in programming to show or hide elements depending on user input, data states, or system status. This technique improves user experience by ensuring that only relevant information is presented at a given time.

What is Condition Rendering in React?

Conditional rendering in React works similarly to conditions in JavaScript. It enables components to display different outputs depending on states or props. This ensures that the UI updates dynamically based on logic instead of manually manipulating the DOM.



Think of it like: "If the user is logged in, show the Dashboard. Otherwise, show the Login page."

Ways to Implement Conditional Rendering in React

Here we will discuss the 5 ways to implement the condition rendering in react:

1. Using If/Else Statements
2. Using Ternary Operator
3. Using Logical AND (&&) Operator
4. Using Function for conditional UI
5. Using Switch Case Statements
6. Conditional Rendering in Lists (Using `.map()`)

React Conditional Rendering

1. Using If/Else Statements (if...else)

If/else statements allow rendering different components based on conditions. This approach is useful for complex conditions.

Step#01: Create a file -> src/components/conditionalRendering/IfElseExample.tsx

Step#02: Add Code

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

function IfElseExample () {
  const [isLoggedIn, setIsLoggedIn] = useState<boolean>(false);

  let content;

  if (isLoggedIn) {
    content = <p>Welcome back, user! </p>;
  } else {
    content = <p>Please log in. </p>;
  }

  return (
    <div>
      <h2>If-Else Example </h2>
      {content}
      <button onClick={() => setIsLoggedIn (!isLoggedIn)}>
        {isLoggedIn ? 'Logout' : 'Login'}
      </button>
    </div>
  );
}

export default IfElseExample;
```

React Conditional Rendering

2. Using Ternary Operator

The ternary operator (condition ? expr1: expr2) is a concise way to conditionally render JSX elements. It's often used when the logic is simple and there are only two options to render.

Step#01: Create a file -> src/components/conditionalRendering/TernaryExample.tsx

Step#02: Add Code

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

function TernaryExample () {
  const [isDarkMode, setIsDarkMode] = useState<boolean>(false);

  return (
    <div style={{ backgroundColor: isDarkMode ? '#222' : '#fff', color: isDarkMode ? '#fff' : '#000', padding: '20px' }}>
      <h2>Ternary Operator Example </h2>
      <p>{isDarkMode ? 'Dark Mode' : 'Light Mode'}</p>
      <button onClick={() => setIsDarkMode (!isDarkMode)}>
        Toggle Mode
      </button>
    </div>
  );
}
```

React Conditional Rendering

3. Using Logical AND (&&)

The && operator returns the second operand if the first is true, and nothing if the first is false. This can be useful when you only want to render something when a condition is true.

Step#01: Create a file -> src/components/conditionalRendering/AndExample.tsx

Step#02: Add Code

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

function AndExample () {
  const [hasNotifications, setHasNotifications] = useState<boolean>(true);

  return (
    <div>
      <h2>Logical AND Example </h2>
      <button onClick={() => setHasNotifications ((prev) => !prev)}>
        Toggle Notifications
      </button>
      {hasNotifications && <p>🔔 You have new notifications! </p>}
    </div>
  );
}

export default AndExample;
```

React Conditional Rendering

4. Using Functions for Conditional UI

The function returns the content based on some condition. This can be useful when we want Reusable, readable logic.

Step#01: Create a file -> src/components/conditionalRendering/FunctionExample.tsx

Step#02: Add Code

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

function FunctionExample() {
  const [role, setRole] = useState<'admin' | 'user' | 'guest'>('guest');

  const renderContent = () => {
    if (role === 'admin') return <p>Welcome, Admin 👑</p>;
    if (role === 'user') return <p>Hello, User 🙌</p>;
    return <p>Welcome, Guest! Please sign up. 🙋</p>;
  };

  return (
    <div>
      <h2>Function Conditional Example</h2>
      {renderContent()}
      <br />
      <button onClick={() => setRole('admin')}>Set Admin</button>
      <button onClick={() => setRole('user')}>Set User</button>
      <button onClick={() => setRole('guest')}>Set Guest</button>
    </div>
  );
}

export default FunctionExample;
```


React Conditional Rendering

5. Using switch-case in function

The function returns the content based on the switch case conditions. More readable for 3+ condition.

Step#01: Create a file -> touch src/components/conditionalRendering/SwitchExample.tsx

Step#02: Add Code

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

function SwitchExample () {
  const [status, setStatus] = useState<'loading' | 'success' | 'error'>('loading');

  const renderStatusMessage = () => {
    switch (status) {
      case 'loading':
        return <p>⏳ Loading... </p>;
      case 'success':
        return <p>✅ Data loaded successfully! </p>;
      case 'error':
        return <p>❌ Something went wrong! </p>;
      default:
        return <p>Unknown status </p>;
    }
  };

  return (
    <div>
      <h2>Switch Case Example </h2>
      {renderStatusMessage ()}
      <div style={{ marginTop: '10px' }}>
        <button onClick={() => setStatus ('loading')}>Loading</button>
        <button onClick={() => setStatus ('success')}>Success</button>
        <button onClick={() => setStatus ('error')}>Error</button>
      </div>
    </div>
  );
}

export default SwitchExample;
```



React Conditional Rendering

6. Conditional Rendering in Lists (Using .map())

Conditional rendering can be helpful when rendering lists of items conditionally. You can filter or map over an array to selectively render components based on a condition.

Step#01: Create a file -> touch src/components/conditionalRendering/ConditionInListExample.tsx

Step#02: Add Code

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

function ConditionInListExample () {
  const items = ["Apple", "Banana", "Cherry"];
  const fruitList = items.map((item, index) =>
    item.includes("a") ? <p key={index}>{item}</p> : null
  );
  return (
    <div>
      <h2>Condition Rendering in List Example </h2>
      <div style={{ marginTop: '10px' }}>
        { fruitList }
      </div>
    </div>
  );
}

export default ConditionInListExample ;
```



React Conditional Rendering

Summary of Conditional Rendering Techniques

Technique	Best for	Inline	Clean Code ?
<code>if...else</code>	Complex logic before render	✗	✓
<code>? :</code> (Ternary)	Toggle between two UI Components	✓	✓
<code>&&</code>	Render if true only	✓	✓
<code>function()</code>	Reusable, Readable logic	✗	✓
<code>switch-case</code>	3+ condition flows	✗	✓



React Conditional Rendering

Practical Use Cases for Conditional Rendering

1. Displaying User Profile Based on Authentication

You can conditionally render a user's profile page if the user is logged in, or a login form if not.

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

function UserProfile() {
  const [isAuthenticated, setIsAuthenticated] = useState(false);

  return (
    <div>
      {isAuthenticated ? (
        <h1>User Profile</h1>
      ) : (
        <button onClick={() =>
          setIsAuthenticated(true)}>Log In</button>
      )}
    </div>
  );
}

export default UserProfile;
```



React Conditional Rendering

Practical Use Cases for Conditional Rendering

Explanation of Example 1

- The component uses **useState** to manage the **isAuthenticated** state, which is initially set to false.
- The **isAuthenticated** state determines what is rendered: if false, a “Log In” button is displayed; if true, a “User Profile” heading appears.
- When the “Log In” button is clicked, **setIsAuthenticated(true)** updates the state to true, triggering a re-render.
- After the state update, the button is replaced with “User Profile”, demonstrating conditional rendering based on authentication state.



React Conditional Rendering

Practical Use Cases for Conditional Rendering

2. Showing Loading State

You can display a loading spinner or message while waiting for data to be fetched.

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

function LoadingDataState () {
  const [isLoading, setIsLoading] = useState(true);
  const [data, setData] = useState(null);

  useEffect(() => {
    setTimeout(() => {
      setData('Fetched Data');
      setIsLoading(false);
    }, 2000);
  }, []);

  return (
    <div>
      {isLoading ? (
        <h1>Loading...</h1>
      ) : (
        <h1>{data}</h1>
      ) }
    </div>
  );
}

export default LoadingDataState;
```



React Conditional Rendering

Practical Use Cases for Conditional Rendering

Explanation of Example 2

- The component uses **useState** to manage two state variables: **isLoading** (initially true) and **data** (initially null).
- The **useEffect** hook runs once after the component mounts and starts a **setTimeout** to simulate fetching data after 2 seconds.
- When the timeout completes, **setData('Fetched Data')** updates the data state, and **setIsLoading(false)** changes **isLoading** to false.
- The component initially renders "Loading...", and once the state updates, it re-renders to display "Fetched Data".



React Hooks

What are Hooks in React?

React Hooks are functions that allow functional components in React to manage state, handle side effects, and access other React features without needing class components. They provide a simpler and more efficient way to manage component logic.

Why Use React Hooks?

- **Simplifies Code:** Hooks provide a simpler and cleaner way to write components by using functions instead of classes.
- **State and Side Effects:** Hooks allow you to use state (`useState`) and side effects (`useEffect`) in functional components.
- **Reusability:** Hooks make it easier to share logic across components by creating custom hooks.
- **Readability:** Functional components with hooks tend to be more concise and easier to read than class components.

Types of React Hooks?

React offers various hooks to handle state, side effects, and other functionalities in functional components. Below are some of the most commonly used types of React hooks:

React Hooks

Types of React Hooks

1. `useState` – for state management

The **`useState`** hook is used to declare state variables in functional components. It allows us to read and update the state within the component.

Step#01: Create a file -> `src/components/reactHooks/UseStateExample.tsx`

Step#02: Add Code

```
import React, { useState, useEffect } from 'react';
function LoadingDataState () {
  const [isLoading, setIsLoading] = useState(true);
  const [data, setData] = useState(null);

  useEffect(() => {
    setTimeout(() => {
      setData('Fetched Data');
      setIsLoading(false);
    }, 2000);
  }, []);

  return (
    <div>
      {isLoading ? (
        <h3>Loading...</h3>
      ) : (
        <h3>{data}</h3>
      ) }
    </div>
  );
}
export default LoadingDataState ;
```



React Hooks

Types of React Hooks

2. `useEffect`— for side effects

The **useEffect** hook in React is used to handle side effects in functional components. It allows you to perform actions such as data fetching, DOM manipulation, and setting up subscriptions, which are typically handled in lifecycle methods like `componentDidMount` or `componentDidUpdate` in class components.

Step#01: Create a file -> `src/components/reactHooks/UseEffectExample.tsx`

Step#02: Add Code

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

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

  useEffect(() => {
    document.title = `Count: ${count}`;
    console.log("Effect ran. Count is: ${count}");

    return () => {
      console.log("Cleanup for previous effect");
      document.title = "React App";
    };
  }, [count]);

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

export default App;
```

React Hooks

Types of React Hooks

3. `useContext` – for Global State Management

Sharing data like theme, user, auth, language, etc.

Avoid prop drilling (passing props down through multiple layers)

Step#01: Create a Context file -> `src/components/reactHooks/UserContext.tsx`

Step#02: Add Code

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

// Step 1: Define types
type UserContextType = {
  loggedIn: boolean;
  toggleLogin: () => void;
};

// Step 2: Create context
const UserContext = createContext<UserContextType | undefined>(undefined);

// Step 3: Create provider component
export const UserProvider = ({ children }: { children: React.ReactNode }) => {
  const [loggedIn, setLoggedIn] = useState(false);

  const toggleLogin = () => setLoggedIn(prev => !prev);

  return (
    <UserContext.Provider value={{ loggedIn, toggleLogin }}>
      {children}
    </UserContext.Provider>
  );
};
```



React Hooks

Types of React Hooks

3. `useContext` – for Global State Management

Step#02: Add Code Continue...

```
// Step 4: Custom hook for easy access
export const useUser = () => {
  const context = useContext(UserContext);
  if (!context) {
    throw new Error('useUser must be used within a UserProvide!');
  }
  return context;
};
```

Step#03: Create **Header.tsx** to consume the context

```
import { useUser } from "../UserContext";

const Header = () => {
  const { loggedIn } = useUser();

  return <h2>{loggedIn ? 'Welcome Back!' : 'Please Log In'}</h2>;
};

export default Header;
```



React Hooks

Types of React Hooks

3. `useContext` – for Global State Management

Step#04: Create **ToggleButton.tsx** to update context

Add Code

```
import { useUser } from "../UserContext";

const ToggleButton = () => {
  const { loggedIn, toggleLogin } = useUser();

  return (
    <button onClick={toggleLogin}>
      {loggedIn ? 'Logout' : 'Login'}
    </button>
  );
};

export default ToggleButton;
```



React Hooks

Types of React Hooks

3. `useContext` – for Global State Management

Step#05: Wrap your app with `UserProvider`

Step#06: Update code in `App.tsx`

```
import Header from './components/reactHooks/Header';
import { UserProvider } from './components/reactHooks/UserContext';
import ToggleButton from './components/reactHooks/ToggleButton';

function App() {
  return (
    <UserProvider>
      <Header />
      <ToggleButton />
    </UserProvider>
  );
}
```



Creating forms in React

Step# 01: Create the Component File by using below command or you can manually create by right click on `src/components`

Path: `src/components/forms/UserForm.tsx`

`echo.> src\components\forms\UserForm.tsx`

Step# 02: Implement the Form Component

Code for **UserForm.tsx** file

```
// src/components/forms/UserForm.tsx
import { useState, ChangeEvent, FormEvent } from 'react';

type FormData = {
  fullName: string;
  email: string;
  password: string;
};

function UserForm(): JSX.Element {
  const [formData, setFormData] = useState<FormData>({
    fullName: '',
    email: '',
    password: '',
  });
}
```



Creating forms in React

Step# 02: Implement the Form Component

```
const [submitted, setSubmitted] = useState<boolean>(false);

// Handle input change
const handleChange = (e: ChangeEvent<HTMLInputElement>) => {
  const { name, value } = e.target;
  setFormData((prevData) => ({
    ...prevData,
    [name]: value,
  }));
};

// Handle form submission
const handleSubmit = (e: FormEvent) => {
  e.preventDefault();

  if (!formData.fullName || !formData.email || !formData.password) {
    alert('Please fill all fields');
    return;
  }

  console.log('Form submitted:', formData);
  setSubmitted(true);
};
```


Creating forms in React

Step# 02: Implement the Form Component

```
return (  
  <div style={{ padding: '20px' }}>  
    <h2>User Registration Form </h2>  
  
    {submitted ? (  
      <p>🎉 Thank you for registering, {formData.fullName}!</p>  
    ) : (  
      <form onSubmit={handleSubmit}>  
        <div>  
          <label htmlFor="fullName">Full Name:</label><br />  
          <input  
            type="text"  
            id="fullName"  
            name="fullName"  
            value={formData.fullName}  
            onChange={handleChange}  
            placeholder="John Doe"  
          />  
        </div>  
  
        <div>  
          <label htmlFor="email">Email:</label><br />  
          <input  
            type="email"  
            id="email"  
            name="email"  
            value={formData.email}  
            onChange={handleChange}  
            placeholder="john@example.com"  
          />  
        </div>  
  
        <div>  
          <label htmlFor="password">Password:</label><br />  
          <input  
            type="password"  
            id="password"  
            name="password"  
            value={formData.password}  
            onChange={handleChange}  
            placeholder="*****"  
          />  
        </div>  
  
        <button type="submit">Register </button>  
      </form>  
    ) }  
  </div>  
);  
}  
  
export default UserForm;
```



Creating forms in React

Step# 03: Use in App.tsx

```
import { useState } from 'react'
import reactLogo from './assets/react.svg'
import viteLogo from '/vite.svg'
import './App.css'
import UserForm from './components/UserForm';
```

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

  return (
    <>
      <UserForm />
    </>
  )
}
```

```
export default App
```



Creating forms in React

Step by Step Explanation

useState<FormData>: We initialize form fields with TypeScript typing

handleChange: Updates state on each keystroke

handleSubmit: Prevents page refresh, validates fields, submits data

submitted: Shows a thank-you message after submission

Routing With React Router





Routing With React Router

React Router Explanation?

One of React's coolest features is undoubtedly the React router. It is a well-known library, so it is required of everyone learning React to be familiar with how to use it. Users must be able to read different pages on a website without difficulty or move between them when using it. The library that makes this possible is React Router.

React builds single-page applications, and the React router is crucial for displaying many views without requiring the browser to reload.

It is important to note that without React Router, it is nearly impossible to display many views in React single-page applications.



Routing With React Router

What is Routing?

The method of routing involves sending users to various websites in response to their requests or actions. The major application for React Js Router is the creation of single page web applications. Multiple routes are defined in the application using the React Router. A user will be sent to a specific route when they enter a particular URL into their browser and that URL path matches any of the “routes” in the router file.

What is React Router DOM?

You may incorporate dynamic routing in a web application development with the help of the npm package React Router DOM. You can do this to show pages and let users traverse them. It is a client- and server-side routing library for React that is completely functional. React Router Dom is used to create single page apps, or programmes that have numerous pages or components but never refresh the page because the content is always fetched dynamically from the URL. With the aid of React Router Dom, this procedure, referred to as routing, is made possible.



Routing With React Router

How to Install a React Router?

Using NPM: Open your terminal, navigate to your project directory, and run the following command:

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

Using Yarn: If you're using yarn as your package manager, run the following command in your terminal within your project directory.

```
$ yarn add react-router-dom
```

Post Installation:

After successfully installing react-router-dom, you can start using it in your React project by importing the necessary components from the package, such as BrowserRouter, Route, Switch, and Link.



Routing With React Router

Navigating between pages (Link, NavLink, useNavigate)

Step#01: Create a New React + TypeScript Project

```
npm create vite@latest my-router-app -- --template react-ts  
cd my-router-app  
npm install
```

Step# 02: Install React Router Dom

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

Step# 03: Setup Folder Structure

Inside **src/**, create folders pages, layouts, components

```
src/  
  pages/  
    Home.tsx  
    About.tsx  
    Dashboard.tsx  
    Profile.tsx  
    Login.tsx  
    NotFound.tsx  
  layouts/  
    DashboardLayout.tsx  
  components/  
    Navbar.tsx
```




Routing With React Router

Navigating between pages (Link, NavLink, useNavigate)

Step#03 - Explanation:

- **pages/** will contain each screen page.
- **layouts/** will contain layout wrapper.
- **components/** will contain reusable UI components like Navbar.

Step# 04: Create All Files One by One

File:  **src/pages/Home.tsx**

```
import React from 'react';

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

export default Home;
```



Routing With React Router

Navigating between pages (Link, NavLink, useNavigate)

File:  src/pages/About.tsx

```
import React from 'react';

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

export default About;
```

- We created two files in above step **Home.tsx** and **About.tsx** similar to this files add few more files in page folder: **Dashboard.tsx**, **Login.tsx**, **Profile.tsx**, **NotFound.tsx**
- Add new file in **components** with name '**NavBar.tsx**'
- Add new file in **layout** folder with name '**DashboardLayout.tsx**'



Routing With React Router

Navigating between pages (Link, NavLink, useNavigate)

File:  `src/components/Navbar.tsx`

Add code in NavBar file to render the menu:

```
import React from 'react';
import { NavLink } from 'react-router-dom';

const Navbar: React.FC = () => {
  return (
    <nav style={{ marginBottom: '20px' }}>
      <NavLink to="/" style={{ margin: '0 10px' }}>Home</NavLink>
      <NavLink to="/About" style={{ margin: '0 10px' }}>About</NavLink>
      <NavLink to="/Dashboard" style={{ margin: '0 10px' }}>Dashboard</NavLink>
    </nav>
  );
};

export default Navbar;
```



Routing With React Router

Navigating between pages (Link, NavLink, useNavigate)

Step# 05: Setup Router in App.tsx

```
import { useState } from 'react';
import { Routes, Route, BrowserRouter, Navigate } from 'react-router-dom';
import Home from './pages/Home';
import About from './pages/About';
import Dashboard from './pages/Dashboard';
import Navbar from './components/Navbar';

function App() {
  return (
    <div>
      <Navbar />
      <Routes>
        <Route path="/" element={<Navigate to="/home" replace />} />
        <Route path="/home" element={<Home />} />
        <Route path="/about" element={<About />} />
        <Route path="/dashboard" element={<Dashboard />} />
      </Routes>
    </div>
  )
}

export default App;
```

Routing With React Router

Navigating between pages (Link, NavLink, useNavigate)

Step# 06: Setup `BrowserRouter` in main.tsx

Wrap your app with **BrowserRouter** tag to enable the routing in the application.

```
1 import { StrictMode } from 'react'
2 import { createRoot } from 'react-dom/client'
3 import './index.css'
4 import App from './App'
5 import { BrowserRouter } from 'react-router-dom';
6
7 createRoot(document.getElementById('root')!).render(

8   <StrictMode>
9     <BrowserRouter>
10       <App />
11     </BrowserRouter>
12   </StrictMode>,
13 )


```



Routing With React Router

Navigating between pages (Link, NavLink, useNavigate)

Explanation of App.tsx

- `/` goes to Home
- `/about` goes to About
- `/dashboard` loads the Dashboard layout and shows nested routes
- `*` catches wrong URLs

Final Important Commands:

Command	Purpose
<code>npm run dev</code>	Start local server
<code>npm install</code>	Install dependencies
<code>npm install react-router-dom</code>	Install router



Routing With React Router

Navigating between pages (Link, NavLink, useNavigate)

1. **Link** - Simple navigation without page reload

Code:

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

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

Use **<Navbar />** inside App.tsx above the routes as was used in previous example.

✓ When you click, it changes the URL and loads component instantly without refreshing.



Routing With React Router

Navigating between pages (Link, NavLink, useNavigate)

2. **NavLink** - Adds automatic "active" style to current link

Code:

```
import { NavLink } from 'react-router-dom';

function Navbar() {
  return (
    <nav>
      <NavLink to="/" style={({ isActive }) => ({ color: isActive ? 'red' : 'black' })}>
        Home
      </NavLink> |
      <NavLink to="/about" style={({ isActive }) => ({ color: isActive ? 'red' : 'black' })}>
        About
      </NavLink>
    </nav>
  );
}
export default Navbar;
```

✓ Useful for active menu highlighting.



Routing With React Router

Navigating between pages (Link, NavLink, useNavigate)

3. `useNavigate` - Programmatic Navigation

Example: Useful when you want to redirect a user after some logic (like after login) or on button click.

Code:

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

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

  const goToProfile = () => {
    navigate('/profile');
  };

  return <button onClick={goToProfile}>Go to Profile</button>;
}
export default Dashboard;
```

✓ Programmatic navigation based on action.



Routing With React Router

Route Parameters (Dynamic Routing) and Query Strings

Dynamic routing lets you capture part of the URL as a variable (like `/user/:id`).

Query strings are extra data attached to the URL after `?`, like `/search?keyword=react`.

Dynamic Params:

Define a route like:

```
<Route path="/user/:id" element={<UserProfile />} />
```

Use `useParams()` to access:

```
const { id } = useParams();
```

✓ Visiting `/user/42` shows the `id = 42`.

Query Strings:

Use `useLocation()`:

```
const { search } = useLocation();  
const params = new URLSearchParams(search);  
const keyword = params.get('keyword');
```

✓ Visiting `/search?keyword=react` will extract `"react"`.



Routing With React Router

Nested Routes

Sometimes you want a layout (like sidebar, navbar) common for multiple child pages. Nested Routes let you group pages under a parent layout.

Practical Steps:

Create a Parent Component with `<Outlet />`:

```
function DashboardLayout() {  
  return (  
    <div>  
      <Navbar />  
      <Outlet />  
    </div>  
  );  
}
```

`<Outlet />` renders the child routes.

Define Routes:

```
<Route path="/dashboard" element={<DashboardLayout />}>  
  <Route path="analytics" element={<Analytics />} />  
  <Route path="settings" element={<Settings />} />  
</Route>.
```



Now `/dashboard/analytics` will render inside `DashboardLayout`.



Routing With React Router

Handling 404 (Not Found) Pages

If user tries to visit a non-existing URL, we must catch it and show a friendly **404 Page Not Found** instead of breaking the app.

Practical Steps:

Create a NotFound Component:

```
function NotFound() {  
  return <h1>404 - Sorry! Page Not Found.</h1>;  
}
```

Catch All Unknown Routes::

```
{/* Catch All */}  
<Route path="*" element={<NotFound />} />
```

✓ * wildcard matches any URL that doesn't exist.