UNIT- III

REACT JS

INTRODUCTION

React JS is a JavaScript library (not a full framework) that's mainly used for building user interfaces — especially for **single-page applications** where the page doesn't reload when you interact with it.

In simple words:

React helps you build dynamic websites that can update parts of the page without refreshing the whole page.

Some key points about React JS:

- **Developed by Facebook** (now Meta).
- **Component-based** — You build small pieces (called components) like buttons, forms, navbars, etc., and combine them to make a full app.
- **Very fast** — Uses something called a **Virtual DOM** to make updates super quick.
- **Reusable code** — Once you create a component, you can reuse it wherever you want.
- **Works with HTML & JavaScript together** (something called **JSX**).

Example: Without React → Every time you click a button, the whole page might reload.

With React → Only that button or part of the page updates instantly without reloading.

When people say **"React is a library, not a full framework,"** they mean:

- **React** only **handles the "view" layer** (UI part) — basically updating and rendering your components efficiently.

View	User Interface (UI)	What the user sees (buttons, forms, pages)

UNIT- III

REACT JS

What is a Single-Page Application (SPA)?

A **Single-Page Application (SPA)** is a web app **that loads only one HTML page** at the start, and **dynamically updates** the page **without reloading the entire page** from the server.

Example:

Traditional Website (Multi-Page)	Single-Page Application (SPA)
Every time you click a link, the whole page reloads from server	Page never reloads. Only parts of page change
Slower because full HTML loads again and again	Faster because only small parts update
Example: Old websites (PHP, WordPress)	Example: Gmail, Facebook, Instagram (React, Vue, Angular apps)

What is Virtual DOM?

✅**Virtual DOM** is a **copy of the real DOM** (the web page structure) but **inside memory**.
✅React uses this **Virtual DOM** to make the web page updates **faster and smoother**.

In Technical Words:

- When something changes (like clicking a button),
- React **does NOT update the real webpage directly**.
- It **updates the Virtual DOM first** (which is very fast).
- Then it **compares** (diffs) the old Virtual DOM with the new one.
- React **finds only the parts that changed**.
- React then **updates ONLY those parts** in the Real DOM!

UNIT- III

REACT JS

✅**No full page reload**

✅**Only tiny updates**

✅**Super fast performance**

In One Line:

Virtual DOM = A smart memory copy of the real page to make changes faster and efficient.

Templating using JSX

What is JSX?

JSX stands for **JavaScript XML**.

It **looks like HTML**, but it's actually **JavaScript** under the hood. You use JSX to **design how your UI should look** in React.

✅JSX lets you **write HTML inside JavaScript** easily.

Why use JSX?

- Easier to **visualize** your UI while coding.
- Cleaner and **less confusing** than building UI using pure JavaScript.
- It **compiles** into simple React.createElement() behind the scenes.

Basic JSX Example

const element = <h1>Hello, world!</h1>;

- <h1>Hello, world!</h1> looks like HTML, but it's **JSX**.
- element is a normal **JavaScript variable** holding that JSX.

UNIT- III

REACT JS

JSX Full Details

1. Multiple Elements Must Be Wrapped

You **cannot return two sibling elements directly** in JSX.

✗This will cause an error:

```
return <h1>Hello</h1><p>World</p>;
```

✓You must wrap them inside a **single parent**, like a or a **React Fragment**:

```
return (
  <div>
    <h1>Hello</h1>
    <p>World</p>
  </div>
);
```

Or using **Fragment**:

```
import React from 'react';
return (
  <>
    <h1>Hello</h1>
    <p>World</p>
  </>
);
```

2. Embedding JavaScript inside JSX

Use **curly braces {}** to run JavaScript inside JSX.

Example:

```
const name = 'John';
return <h1>Hello, {name}!</h1>;
```

It will show: Hello, John!

You can even do calculations inside:

```
return <p>2 + 2 = {2 + 2}</p>;
```

Output: 2 + 2 = 4

3. JSX Attributes

Attributes in JSX **look like HTML**, but some are **camelCase**.

HTML Attribute	JSX Attribute
class	className
for	htmlFor

Example:

```
return <h1 className="title">Hello World</h1>;
```

4. Inline Styling in JSX

To apply CSS **directly** using JSX, you pass a **JavaScript object** inside {}.

Example:

```
const style = {
  color: 'blue',
  backgroundColor: 'yellow',
```

```
  fontSize: '20px'
};

return <h1 style={style}>Styled Text</h1>;
```

or directly inside:

```
return <h1 style={{ color: 'red', fontWeight: 'bold' }}>Hello</h1>;
```

5. Conditional Rendering (if/else in JSX)

JSX doesn't allow if/else directly.
Instead, you use **ternary operators** or **short-circuiting**.

Example using ternary:

```
const isLoggedIn = true;
return (
  <div>
    {isLoggedIn ? <p>Welcome back!</p> : <p>Please login.</p>}
  </div>
);
```

Example using &&:

```
const isAdmin = true;
return (
  <div>
    {isAdmin && <p>Admin Panel</p>}
  </div>
);
```

UNIT- III

REACT JS

6. Lists in JSX (Looping)

You can use .map() to render lists.

Example:

```
const users = ['John', 'Jane', 'Alice'];
return (
  <ul>
    {users.map((user, index) => (
      <li key={index}>{user}</li>
    ))}
  </ul>
);
```

- Always add a **unique key prop** when rendering lists!

JSX Final Notes:

Feature	Description
Curly Braces {}	To embed JS
className	Instead of class
style Object	Inline CSS styling
Ternary & &&	Conditional rendering
map()	To render lists
Must wrap in parent	div or Fragment

In Short:

JSX = **Write HTML in JavaScript** + **Add Power of JavaScript in HTML**

UNIT- III

REACT JS

Example Program

Profile Card

```
import React from 'react';

function ProfileCard() {
  const user = {
    name: 'John Doe',
    age: 24,
    bio: 'A passionate developer ',
    profilePicture: 'https://randomuser.me/api/portraits/men/75.jpg'
  };
  const cardStyle = {
    border: '1px solid #ccc',
    borderRadius: '10px',
    width: '300px',
    padding: '20px',
    textAlign: 'center',
    boxShadow: '0px 0px 10px rgba(0,0,0,0.1)'
  };
  const imageStyle = {
    width: '100px',
    height: '100px',
    borderRadius: '50%',
```

```
    objectFit: 'cover',
    marginBottom: '10px'
  };
  return (
    <div style={cardStyle}>
      <img src={user.profilePicture} alt="Profile" style={imageStyle} />
      <h2>{user.name}</h2>
      <p>Age: {user.age}</p>
      <p>{user.bio}</p>
    </div>
  );
}
export default ProfileCard;
```

What this JSX is doing:

Part	What's Happening
{user.name}	Embeds JavaScript value into the HTML
{} inside style	Inline CSS Styling
	Dynamic Image
Whole layout inside	Wrapped in a single parent

How to Render it:

In your index.js:

```
import React from 'react';
```

UNIT- III

REACT JS

```
import ReactDOM from 'react-dom';

import ProfileCard from './ProfileCard';

ReactDOM.render(

  <ProfileCard />,

  document.getElementById('root')

);
```

How it will look visually:

[Profile Picture]

John Doe

Age: 24

A passionate developer

ReactJS Components

In **ReactJS**, components are the **building blocks** that help you create **interactive UIs** by dividing your app into smaller, **reusable pieces of code**. Understanding how **components** work is essential for efficiently developing **React applications**.

What are ReactJS Components?

Components in React are JavaScript functions or classes that return a piece of UI. These components allow developers to build complex UIs from small, isolated, and reusable pieces. React components are the core building blocks for any React application and can manage their state, handle user inputs, and render dynamic content.

Types of React Components

There are two main types of components in React:

- **Functional Components**
- **Class Components**

UNIT- III

REACT JS

Functional Components

A Functional Component is a simpler and more concise way of writing components in React using JavaScript functions. These components receive props (properties) as an argument and return JSX (JavaScript XML) to define the UI structure.

```
import React, { useState } from 'react';
const Welcome = () => {
  const [message, setMessage] = useState("Hello, World!");
  return (
    <div>
      <h1>{message}</h1>
      <button onClick={() => setMessage("Hello, React!")}>
        Change Message
      </button>
    </div>
  );
};
export default Welcome;
```

Output

Functional Components

In this code

- useState is used to manage the message state, initially set to "Hello, World!".

- The button click triggers setMessage, which updates the message state to "Hello, React!".
- The component displays the message in an <h1> element and updates it when the button is clicked.

Class Components

Class components in React are ES6 classes that extend the React.Component class. They are used for creating components that need to have their own state or lifecycle methods. While functional components are now the go-to choice for many developers (especially with the introduction of hooks like useState and useEffect), class components still have their place and provide a more traditional way of handling component logic in React.

```
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 });
  };
  decrement = () => {
    this.setState({ count: this.state.count - 1 });
  };
  render() {
    return (
      <div>
        <h1>Counter: {this.state.count}</h1>
        <button onClick={this.increment}>Increment</button>
        <button onClick={this.decrement}>Decrement</button>
      </div>
```

```
      );
   }
}

export default Counter;
```

Output

Class Components in React

In this code

- **state:** The counter (count) is initialized to 0 in the constructor.
- **increment and decrement:** These methods update the count state by 1 when the respective buttons are clicked.
- **render:** Displays the current count and two buttons to increment or decrement the counter.

Difference between Class component and Functional component

Feature	Functional Components	Class Components
Definition	A function that returns JSX.	A class that extends React.Component and has a render() method.
State	Can use state via hooks like useState.	Can manage state using this.state and this.setState().

UNIT- III

REACT JS

Feature	Functional Components	Class Components
Lifecycle Methods	No lifecycle methods (use useEffect hook).	Uses lifecycle methods like componentDidMount, componentDidUpdate, componentWillUnmount.
Syntax	Simpler syntax, function-based.	More verbose, class-based syntax.
Performance	More lightweight and performant.	Slightly heavier due to the class structure and additional methods.
Use of Hooks	Can use hooks like useState, useEffect, etc.	Cannot use hooks directly.
Rendering	Directly returns JSX.	Must define a render() method to return JSX.
Context	Easier to integrate with React hooks like useContext.	Uses Context.Consumer for context integration.

Functional Component (with Hooks)

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

// Creating Context for global state

const CountContext = createContext();

const Counter = () => {
```

UNIT- III

REACT JS

```
// useState Hook: to manage the count state

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

// useRef Hook: to focus on the input element

const inputRef = useRef(null);

// useEffect Hook: to run code when the component mounts

useEffect(() => {

  console.log('Component Mounted or Updated');

  // Focusing on the input element after render

  inputRef.current.focus();

  return () => {

    console.log('Component Unmounted');

  };

}, [count]); // Dependency array ensures it runs when count changes

// useContext Hook: Accessing the value from the context

const globalCount = useContext(CountContext);

return (

  <div>

    <h1>Functional Component - Counter</h1>
```

```
      <p>Current Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
      <button onClick={() => setCount(count - 1)}>Decrement</button>
      {/* Using useRef to focus the input */}
      <input ref={inputRef} type="text" placeholder="Focus me" />
      {/* Displaying context value */}
      <p>Global Count from Context: {globalCount}</p>
    </div>
  );
};
// App Component with Context.Provider
const App = () => {
  const [globalCount, setGlobalCount] = useState(10);
  return (
    <CountContext.Provider value={globalCount}>
      <Counter />
      <button onClick={() => setGlobalCount(globalCount + 1)}>Increase Global Count</button>
    </CountContext.Provider>
```

```
  );

};

export default App;
```

Explanation of Functional Component with Hooks:

1. **useState**: Manages the local state count. It returns the current state and a function to update it.
2. **useEffect**: Runs side effects like logging messages or focusing the input field. Here, it runs on every render because count is part of the dependency array.
3. **useRef**: Creates a reference to the input field, and after each render, it focuses on the input element.
4. **useContext**: Allows us to consume the value from a CountContext provider. The globalCount value is managed at the App level but accessed inside Counter via useContext.

Class Component (with Lifecycle Methods)

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

// Creating Context for global state

const CountContext = React.createContext();

class Counter extends Component {

  constructor(props) {

    super(props);

    // Initializing state
```

```
  this.state = {
    count: 0,
  };
  this.inputRef = React.createRef(); // For input focus
}
// Lifecycle method: componentDidMount (similar to useEffect with empty dependency array)
componentDidMount() {
  console.log('Component Mounted');
  // Focus on the input element after mounting
  this.inputRef.current.focus();
}
// Lifecycle method: componentDidUpdate (similar to useEffect with count as dependency)
componentDidUpdate(prevProps, prevState) {
  if (prevState.count !== this.state.count) {
    console.log('Count has been updated');
  }
}
// Lifecycle method: componentWillUnmount (similar to cleanup in useEffect)
```

```
componentWillUnmount() {
  console.log('Component will Unmount');
}
render() {
  return (
    <div>
      <h1>Class Component - Counter</h1>
      <p>Current Count: {this.state.count}</p>
      <button onClick={() => this.setState({ count: this.state.count + 1 })}>Increment</button>
      <button onClick={() => this.setState({ count: this.state.count - 1 })}>Decrement</button>
      {/* Using createRef to focus the input */}
      <input ref={this.inputRef} type="text" placeholder="Focus me" />
      {/* Consuming Context value */}
      <CountContext.Consumer>
        {(globalCount) => <p>Global Count from Context: {globalCount}</p>}
      </CountContext.Consumer>
    </div>
  );
```

```
  }
}
// App Component with Context.Provider
class App extends Component {
  constructor(props) {
    super(props);
    this.state = {
      globalCount: 10,
    };
  }
  render() {
    return (
      <CountContext.Provider value={this.state.globalCount}>
        <Counter />
        <button onClick={() => this.setState({ globalCount: this.state.globalCount + 1 })}>Increase Global Count</button>
      </CountContext.Provider>
    );
  }
```

```
}

export default App;
```

Explanation of Class Component with Lifecycle Methods:

1. **State**: Managed through this.state, and updated using this.setState().
2. **Lifecycle Methods**:
   - **componentDidMount()**: Equivalent to useEffect() with an empty dependency array. This is called after the component mounts.
   - **componentDidUpdate()**: Similar to useEffect() that runs when a specific value (count) changes.
   - **componentWillUnmount()**: Equivalent to the cleanup function in useEffect(), called before the component is removed.
3. **createRef**: A reference is created to focus the input element, similar to useRef.
4. **Context**: Consumed via CountContext.Consumer to access the global globalCount.

Comparison of Functional and Class Component:

Feature	Functional Component (with Hooks)	Class Component (with Lifecycle Methods)
State Management	useState()	this.state, this.setState()
Side Effects	useEffect()	componentDidMount(), componentDidUpdate(), componentWillUnmount()
Refs	useRef()	React.createRef()
Context Consumption	useContext()	CountContext.Consumer
Focus Management	useRef() (for focusing input)	React.createRef() (for focusing input)

UNIT- III

REACT JS

Feature	Functional Component (with Hooks)	Class Component (with Lifecycle Methods)
Clean Up	useEffect() clean-up function	componentWillUnmount()

Feature Comparison: Functional Components (with Hooks) vs Class Components (with Lifecycle Methods)

Explanation of Features:

1. State Management:

- **Functional Components**: Use the useState() hook to manage local component state. useState() returns an array where the first value is the state variable, and the second value is a function to update the state.

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

- **Class Components**: Use this.state to define state and this.setState() to update the state. State is usually defined in the constructor, and this.setState() triggers a re-render.

  ```
  constructor(props) {
     super(props);
     this.state = { count: 0 };
  }
  this.setState({ count: 1 });
  ```

2. Side Effects:

- **Functional Components**: useEffect() hook is used to handle side effects like data fetching, subscriptions, or manually changing the DOM. It can also act like multiple lifecycle methods (componentDidMount, componentDidUpdate, and componentWillUnmount) depending on how dependencies are set.

```
useEffect(() => {
   // Side effect code (e.g., fetching data)
}, []); // Empty dependency array means it runs once (componentDidMount)
```

- **Class Components**: The equivalent lifecycle methods are:
  - componentDidMount() — runs once after the component is mounted (similar to useEffect with an empty dependency array).
  - componentDidUpdate() — runs whenever the component updates (similar to useEffect with dependencies).
  - componentWillUnmount() — called before the component is removed from the DOM for cleanup (similar to useEffect cleanup).

3. Refs:

- **Functional Components**: useRef() is used to persist references to DOM elements or mutable values across renders without causing re-renders. It's also used for focus management.

```
const inputRef = useRef(null);
inputRef.current.focus(); // Focus input
```

- **Class Components**: React.createRef() is used to create a reference and attach it to a DOM element.

```
this.inputRef = React.createRef();
this.inputRef.current.focus(); // Focus input
```

4. Context Consumption:

- **Functional Components**: useContext() allows you to consume context values directly without needing to wrap the component in a Context.Consumer.

```
const value = useContext(MyContext);
```

- **Class Components**: You use CountContext.Consumer to consume values from the context. This requires a render prop pattern.

```
<CountContext.Consumer>
  {value => <div>{value}</div>}
</CountContext.Consumer>
```

5. Focus Management:

- **Functional Components**: You can use useRef() to store a reference to an input element and then focus it when necessary.

```
const inputRef = useRef();
inputRef.current.focus();
```

- **Class Components**: Use React.createRef() to manage focus in class components.

```
this.inputRef = React.createRef();
this.inputRef.current.focus();
```

6. Clean Up:

- **Functional Components**: You can return a clean-up function inside useEffect() to handle component cleanup, such as unsubscribing from a service or clearing timers.

```
useEffect(() => {
   const timer = setTimeout(() => console.log('Timer!'), 1000);
   return () => clearTimeout(timer); // Clean up
}, []);
```

- **Class Components**: componentWillUnmount() is used for cleanup, like clearing timers or canceling network requests.

```
componentWillUnmount() {
```

```
    clearTimeout(this.timer);
  }
```

Summary:

- **Functional Components** with **Hooks** provide a more concise, cleaner, and modular way to handle state, side effects, and other features.
- **Class Components** use lifecycle methods and internal state to manage component logic, but they are more verbose and less flexible compared to functional components with hooks.

State and Props in React

In React, **state** and **props** are two essential concepts that help manage and pass data between components. They are used to control how the user interface (UI) looks and behaves. Here's an overview of both:

1. State

State is an object that holds data or information about the component's behavior and can change over time. When the state of a component changes, React automatically re-renders that component to reflect the new state

- **State is local**: Each component can have its own state.
- **State is mutable**: It can be updated during the lifecycle of the component.

In **Functional Components**, you manage state using the useState() hook, while in **Class Components**, you use this.state and this.setState().

Example of State in Functional Component (using useState):

```
import React, { useState } from 'react';
const Counter = () => {
  // Declare state using useState hook
```

```
    const [count, setCount] = useState(0);
    // Function to increment the count
    const increment = () => {
        setCount(count + 1);
    };
    return (
        <div>
            <p>Count: {count}</p>
            <button onClick={increment}>Increment</button>
        </div>
    );
};
export default Counter;
```

In this example:

- The useState(0) hook initializes the count state to 0.
- setCount is the function used to update the state (count).

Example of State in Class Component:

```
import React, { Component } from 'react';
class Counter extends Component {
    constructor(props) {
        super(props);
        // Initialize state in the constructor
        this.state = { count: 0 };
    }
    // Function to increment the count
    increment = () => {
        this.setState({ count: this.state.count + 1 });
    };
```

```
  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <button onClick={this.increment}>Increment</button>
      </div>
    );
  }
}
export default Counter;
```

In this example:

- The this.state is used to store the state (count).
- this.setState() is used to update the state.

2. Props

Props (short for **properties**) are read-only data that are passed from a parent component to a child component. Props are used to pass information, including data and event handlers, between components. A component cannot modify its own props, but it can pass them to other components as needed.

- **Props are immutable**: They cannot be changed by the component receiving them.
- **Props allow communication** between components by sending data down the component tree.

Example of Props in Functional Component:

```
import React from 'react';
const Greeting = (props) => {
  return <h1>Hello, {props.name}!</h1>;
};
```

```
const App = () => {
   return <Greeting name="John" />;
};
export default App;
```

In this example:

- The Greeting component receives the name prop from the parent (App component).
- props.name is used inside the Greeting component to display the name passed from the parent.

Example of Props in Class Component:

```
import React, { Component } from 'react';
class Greeting extends Component {
   render() {
      return <h1>Hello, {this.props.name}!</h1>;
   }
}
class App extends Component {
   render() {
      return <Greeting name="John" />;
   }
}
export default App;
```

In this example:

- The Greeting component receives the name prop from the parent (App component).
- this.props.name is used inside the Greeting component to display the name passed from the parent.

UNIT- III

REACT JS

State vs Props

Feature	State	Props
Definition	Data that is owned and managed by the component.	Data that is passed down from parent to child components.
Mutability	Mutable. Can be changed using setState() (class) or useState() (functional).	Immutable. Cannot be changed by the child component.
Scope	Local to the component where it is defined.	Can be passed to child components, making them global within the component tree.
Use Case	Used for internal state that affects the component's behavior or appearance.	Used to pass data or event handlers from a parent to a child.
Update Trigger	Re-renders the component when state changes.	A change in props will cause the child component to re-render.
Initialization	Initialized within the component itself.	Initialized in the parent component and passed down to the child.

Example: Combining State and Props

In a real-world scenario, we often combine both state and props in a component to manage data and pass it around.

```
import React, { useState } from 'react';
// Child Component
const Message = (props) => {
    return <p>{props.text}</p>;
};
```

UNIT- III

REACT JS

```
// Parent Component
const App = () => {
  const [message, setMessage] = useState("Welcome to React!");
  return (
    <div>
      <Message text={message} />
      <button onClick={() => setMessage("You clicked the button!")}>
        Change Message
      </button>
    </div>
  );
};
export default App;
```

In this example:

- The **parent component (App)** manages the state message using useState().
- The **child component (Message)** receives the message as a prop and displays it.
- Clicking the button in the parent component updates the state (message), which automatically re-renders the parent and child components.

Conclusion

- **State** is used to manage data that can change over time and is local to the component.
- **Props** are used to pass data from a parent component to a child component and are immutable within the child component.

React Component Lifecycle

Class Component Lifecycle Methods

1. **Mounting**:
    - constructor(): Initializes state and binds methods.

- render(): Renders UI.
- componentDidMount(): Called after the component mounts.

2. **Updating**:
   - shouldComponentUpdate(): Determines if a re-render is needed.
   - render(): Re-renders the component.
   - componentDidUpdate(): Called after the component updates.
3. **Unmounting**:
   - componentWillUnmount(): Cleanup before removal from the DOM.
4. **Error Handling**:
   - getDerivedStateFromError(), componentDidCatch(): Handle errors in child components.

Functional Component with Hooks (Using useEffect)

1. **Mounting**:
   - useEffect(() => { ... }, []): Runs once when the component mounts (similar to componentDidMount).
2. **Updating**:
   - useEffect(() => { ... }, [dependency]): Runs when a specific state/prop changes (similar to componentDidUpdate).
3. **Unmounting**:
   - useEffect(() => { return () => { ... }; }, []): Cleanup on unmount (similar to componentWillUnmount).

Comparison Table:

Feature	Class Component	Functional Component
Mounting	componentDidMount()	useEffect(() => { ... }, [])
Updating	shouldComponentUpdate()	useEffect(() => { ... }, [dep])

UNIT- III

REACT JS

Feature	Class Component	Functional Component
Unmounting	componentWillUnmount()	useEffect(() => { return cleanup }, [])
Error Handling	componentDidCatch()	Error boundaries

In summary, **class components** use lifecycle methods to manage state and effects, while **functional components** leverage useEffect for the same purpose with simpler syntax.

Here are simple examples for both **class components** and **functional components** demonstrating the lifecycle methods and hooks:

Class Component Example (Using Lifecycle Methods)

```
import React, { Component } from 'react';
class LifecycleExample extends Component {
  constructor(props) {
    super(props);
    this.state = {
      message: 'Hello, World!',
    };
    console.log('Constructor: Component initialized');
  }
  // Called after the component mounts
  componentDidMount() {
    console.log('componentDidMount: Component mounted');
  }
  // Called before the component updates
  shouldComponentUpdate(nextProps, nextState) {
    console.log('shouldComponentUpdate: Checking if re-render is needed');
    return true; // Allows the re-render
  }
```

```
  // Called after the component updates
  componentDidUpdate(prevProps, prevState) {
    console.log('componentDidUpdate: Component updated');
  }
  // Called before the component unmounts
  componentWillUnmount() {
    console.log('componentWillUnmount: Cleanup before removal');
  }
  render() {
    return (
      <div>
        <h1>{this.state.message}</h1>
        <button
          onClick={() => this.setState({ message: 'Hello, React!' })}
        >
          Change Message
        </button>
      </div>
    );
  }
}
export default LifecycleExample;
```

Explanation:

1. **constructor()** initializes the state.
2. **componentDidMount()** runs once after the component is mounted.
3. **shouldComponentUpdate()** determines whether the component should re-render.
4. **componentDidUpdate()** runs after the component is updated.
5. **componentWillUnmount()** is used for cleanup before the component is removed.

UNIT- III

REACT JS

Functional Component Example (Using useEffect Hook)

```
import React, { useState, useEffect } from 'react';
const LifecycleExample = () => {
  const [message, setMessage] = useState('Hello, World!');
  // ComponentDidMount and ComponentDidUpdate
  useEffect(() => {
    console.log('useEffect: Component mounted or updated');
    // Cleanup function (ComponentWillUnmount)
    return () => {
      console.log('Cleanup: Component will unmount');
    };
  }, [message]); // Runs when 'message' changes
  return (
    <div>
      <h1>{message}</h1>
      <button onClick={() => setMessage('Hello, React!')}>
        Change Message
      </button>
    </div>
  );
};
export default LifecycleExample;
```

Explanation:

1. **useEffect()** is used for mounting and updating effects.
2. It runs after the component is mounted and when the message state changes (like componentDidMount and componentDidUpdate).
3. The **cleanup function** inside useEffect works like componentWillUnmount() and runs before the component is unmounted or when message changes again.

UNIT- III

REACT JS

Comparison:

- **Class Component**: Uses specific lifecycle methods like componentDidMount, componentDidUpdate, etc.
- **Functional Component**: Uses the useEffect hook for managing side effects and cleanup.

1. Rendering List and Portals

In **React.js**, **rendering lists** and using **portals** are common techniques used in component rendering and DOM manipulation.

Rendering a List in React

Rendering lists is typically done using the .map() function.

Example: Rendering a list of users

```
import React from 'react';
const UserList = ({ users }) => {
  return (
    <ul>
      {users.map((user, index) => (
        <li key={user.id || index}>{user.name}</li>
      ))}
    </ul>
  );
};
// Example usage:
const users = [
  { id: 1, name: 'Alice' },
  { id: 2, name: 'Bob' },
];
export default function App() {
```

```
  return <UserList users={users} />;
}
```

Notes:

- Always use a unique key when rendering lists.
- Avoid using index as key unless no unique id is available.

2. Portals in React

Portals allow you to **render children into a DOM node that exists outside the parent component's hierarchy**.

Example: Modal using React Portal

Step 1: Create a DOM node in your public/index.html

```
<!-- Add this outside the root div -->
<div id="modal-root"></div>
```

Step 2: Create the Portal Component

```
import React from 'react';
import ReactDOM from 'react-dom';
const Modal = ({ children }) => {
  return ReactDOM.createPortal(
    <div className="modal">
      {children}
    </div>,
    document.getElementById('modal-root')
  );
};
export default Modal;
```

Step 3: Use it in your app

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

```
import Modal from './Modal';
export default function App() {
  const [showModal, setShowModal] = useState(false);
  return (
    <div>
      <button onClick={() => setShowModal(true)}>Open Modal</button>
      {showModal && (
        <Modal>
          <div>
            <h2>Hello from Portal!</h2>
            <button onClick={() => setShowModal(false)}>Close</button>
          </div>
        </Modal>
      )}
    </div>
  );
}
```

Error Handling

Error handling in **React.js** is important to keep your app stable and user-friendly. React provides several ways to handle errors depending on **where** and **how** they occur:

1. Try-Catch in Event Handlers or Async Functions

For errors in functions like API calls or button clicks:

```
const handleClick = async () => {
  try {
    const response = await fetch('/api/data');
    const data = await response.json();
    console.log(data);
  } catch (error) {
```

```
    console.error('Something went wrong:', error);
    alert('Failed to fetch data!');
  }
};
```

2. Error Boundaries (Class Components Only)

For rendering errors (e.g., a component crashes):

Step 1: Create an Error Boundary Component

```
import React, { Component } from 'react';
class ErrorBoundary extends Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }
  static getDerivedStateFromError() {
    return { hasError: true };
  }
  componentDidCatch(error, info) {
    console.error('ErrorBoundary caught an error', error, info);
  }
  render() {
    if (this.state.hasError) {
      return <h2>Something went wrong □</h2>;
    }
    return this.props.children;
  }
}
export default ErrorBoundary;
```

Step 2: Wrap it around risky components

```
import ErrorBoundary from './ErrorBoundary';
import BuggyComponent from './BuggyComponent';
function App() {
  return (
    <div>
      <ErrorBoundary>
        <BuggyComponent />
      </ErrorBoundary>
    </div>
  );
}
```

3. Handling Errors in Functional Components with Hooks

If you're using hooks, use useEffect() + try-catch or use libraries like react-error-boundary:

Note: npm install react-error-boundary (Install in Terminal)

```
import { ErrorBoundary } from 'react-error-boundary';
function ErrorFallback({ error, resetErrorBoundary }) {
  return (
    <div>
      <p>Something went wrong:</p>
      <pre>{error.message}</pre>
      <button onClick={resetErrorBoundary}>Try again</button>
    </div>
  );
}
function MyComponent() {
  throw new Error('Oops!');
}
```

```
function App() {
  return (
    <ErrorBoundary FallbackComponent={ErrorFallback}>
      <MyComponent />
    </ErrorBoundary>
  );
}
```

4. 404 and Network Errors (Route Not Found)

Use React Router for 404 handling:

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

Handle API/network errors in try-catch as shown earlier.

Summary

Error Type	Handling Method
API / Network Errors	try-catch in async functions
Rendering Errors	Error Boundaries (class or library)
Form Validation Errors	Conditional rendering, e.g. {error && Error }
Route Not Found	React Router path="*" route

Routers

In **React**, routing is managed using a library called **React Router** (react-router-dom) for web apps. It allows navigation between different components/pages without refreshing the whole page—this is known as **client-side routing**.

UNIT- III

REACT JS

Step-by-Step: React Router Setup

1. Install React Router

npm install react-router-dom

2. Basic Routing Example

 App.js

```
import React from 'react';
import { BrowserRouter as Router, Routes, Route, Link } from 'react-router-dom';
import Home from './pages/Home';
import About from './pages/About';
import Contact from './pages/Contact';
import NotFound from './pages/NotFound';
function App() {
  return (
    <Router>
      <nav>
        <Link to="/">Home</Link> |
        <Link to="/about">About</Link> |
        <Link to="/contact">Contact</Link>
      </nav>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
        <Route path="/contact" element={<Contact />} />
        <Route path="*" element={<NotFound />} />
      </Routes>
    </Router>
  );
}
export default App;
```

UNIT- III

REACT JS

3. Create Page Components

□ *pages/Home.js*

```
const Home = () => <h2>□ Home Page</h2>;
export default Home;
```

□ *pages/About.js*

```
const About = () => <h2>ℹ About Us</h2>;
export default About;
```

□ *pages/Contact.js*

```
const Contact = () => <h2>□ Contact Us</h2>;
export default Contact;
```

□ *pages/NotFound.js*

```
const NotFound = () => <h2>❌ 404 Page Not Found</h2>;
export default NotFound;
```

□ Optional: Redirects and Nested Routes

➤ Redirect Example

```
import { Navigate } from 'react-router-dom';
<Route path="/old-contact" element={<Navigate to="/contact" replace />} />
```

➤ Nested Routes Example

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

Use <Outlet /> inside Dashboard.js to render nested components.

UNIT- III

REACT JS

Summary Table

Feature	Usage
Define route paths	<Route path="/path" element={<Comp />} />
Navigate links	<Link to="/about">About</Link>
404 fallback	<Route path="*" element={<NotFound />}/>
Redirect	<Navigate to="/new" replace />
Nested Routes	Use <Outlet /> and child <Route>s

React Router demo project with working routes, navigation, a 404 page, and nested routes

Project Structure

react-router-demo/

├── App.js

├── index.js

└── pages/

├── Home.js

├── About.js

├── Contact.js

├── Dashboard.js

UNIT- III

REACT JS

├── **Profile.js**

├── **Settings.js**

└── **NotFound.js**

index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
ReactDOM.render(<App />, document.getElementById('root'));
```

App.js

```
import React from 'react';
import { BrowserRouter as Router, Routes, Route, Link } from 'react-router-dom';
import Home from './pages/Home';
import About from './pages/About';
import Contact from './pages/Contact';
import Dashboard from './pages/Dashboard';
import Profile from './pages/Profile';
import Settings from './pages/Settings';
import NotFound from './pages/NotFound';
function App() {
  return (
    <Router>
      <nav style={{ padding: '10px', background: '#eee' }}>
        <Link to="/">Home</Link> |
        <Link to="/about"> About</Link> |
        <Link to="/contact"> Contact</Link> |
        <Link to="/dashboard"> Dashboard</Link>
      </nav>
```

```
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
        <Route path="/contact" element={<Contact />} />
        {/* Nested Routes */}
        <Route path="/dashboard" element={<Dashboard />}>
          <Route path="profile" element={<Profile />} />
          <Route path="settings" element={<Settings />} />
        </Route>
        {/* 404 Page */}
        <Route path="*" element={<NotFound />} />
      </Routes>
    </Router>
  );
}
export default App;
```

□ pages/Home.js

```
const Home = () => <h2>□ Welcome to Home Page</h2>;
export default Home;
```

□ pages/About.js

```
const About = () => <h2>ℹ About Us</h2>;
export default About;
```

□ pages/Contact.js

```
const Contact = () => <h2>□ Contact Page</h2>;
export default Contact;
```

□ pages/Dashboard.js

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

```
const Dashboard = () => (
  <div>
    <h2>□ Dashboard</h2>
    <Link to="profile">Profile</Link> | <Link to="settings">Settings</Link>
    <Outlet />
  </div>
);
export default Dashboard;
```

□ pages/Profile.js

```
const Profile = () => <p>□ User Profile Section</p>;
export default Profile;
```

□ pages/Settings.js

```
const Settings = () => <p>□ Account Settings</p>;
export default Settings;
```

□ pages/NotFound.js

```
const NotFound = () => <h2>❌404 - Page Not Found</h2>;
export default NotFound;
```

To Run This Project

npx create-react-app react-router-demo
cd react-router-demo
npm install react-router-dom
Replace the default files with the above and run:
npm start

UNIT- III

REACT JS

Redux and Redux Saga

What is Redux?

Redux is a **state management** library that helps manage and centralize application state. It uses:

- **Store** – Global state container
- **Actions** – Events that describe state changes
- **Reducers** – Functions that update the store based on actions

What is Redux-Saga?

Redux-Saga is a **middleware** library used to handle **asynchronous side effects** (like API calls) in a Redux app using generator functions (function*).

It allows:

- Delayed dispatches
- Asynchronous API requests
- Background processes (e.g., polling, debouncing)

Step-by-Step Setup: Redux + Redux Saga

1. Install Required Packages

npm install redux react-redux redux-saga

2. Project Structure

src/
├── redux/
│ ├── actions.js
│ ├── reducers.js
│ ├── sagas.js

```
│   └── store.js
├── components/
│   └── Counter.js
└── App.js
```

Example Use Case: Counter with Saga Delay

redux/actions.js

```
export const INCREMENT = "INCREMENT";
export const DECREMENT = "DECREMENT";
export const INCREMENT_ASYNC = "INCREMENT_ASYNC";
export const increment = () => ({ type: INCREMENT });
export const decrement = () => ({ type: DECREMENT });
export const incrementAsync = () => ({ type: INCREMENT_ASYNC });
```

redux/reducers.js

```
import { INCREMENT, DECREMENT } from './actions';
const initialState = { count: 0 };
export const counterReducer = (state = initialState, action) => {
  switch (action.type) {
    case INCREMENT: return { count: state.count + 1 };
    case DECREMENT: return { count: state.count - 1 };
    default: return state;
  }
};
```

redux/sagas.js

```
import { put, takeEvery, delay } from 'redux-saga/effects';
import { INCREMENT_ASYNC, increment } from './actions';
function* incrementAsyncSaga() {
  yield delay(1000); // wait 1 sec
  yield put(increment());
```

```
}
export function* rootSaga() {
  yield takeEvery(INCREMENT_ASYNC, incrementAsyncSaga);
}
```

redux/store.js

```
import { createStore, applyMiddleware } from 'redux';
import createSagaMiddleware from 'redux-saga';
import { counterReducer } from './reducers';
import { rootSaga } from './sagas';
const sagaMiddleware = createSagaMiddleware();
export const store = createStore(
  counterReducer,
  applyMiddleware(sagaMiddleware)
);
sagaMiddleware.run(rootSaga);
```

components/Counter.js

```
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { increment, decrement, incrementAsync } from '../redux/actions';
const Counter = () => {
  const count = useSelector(state => state.count);
  const dispatch = useDispatch();
  return (
    <div>
      <h2>Count: {count}</h2>
      <button onClick={() => dispatch(increment())}>+ Increment</button>
      <button onClick={() => dispatch(decrement())}>- Decrement</button>
      <button onClick={() => dispatch(incrementAsync())}>+ Async (Saga)</button>
    </div>
```

```
  );
};
export default Counter;
```

App.js

```
import React from 'react';
import { Provider } from 'react-redux';
import { store } from './redux/store';
import Counter from './components/Counter';
const App = () => (
  <Provider store={store}>
    <h1>Redux + Redux-Saga Demo</h1>
    <Counter />
  </Provider>
);
export default App;
```

Summary

Tool	Purpose
redux	Global state management
react-redux	Connect Redux with React components
redux-saga	Handle async logic via generators

Immutable.js

Immutable.js is a JavaScript library from Facebook designed to create **immutable** data structures like Lists, Maps, Sets, etc. In **React**, it's often used to improve performance and ensure predictable state updates.

UNIT- III

REACT JS

Why Use in React?

1. **Immutability by** Immutable.js **Design**: Prevents accidental mutations of state.
2. **Performance Optimization**: Fast comparisons via === (reference equality).
3. **Cleaner Code**: Chained operations like map(), filter(), etc., are supported out of the box.
4. **Easy Undo/Redo Logic**: Track state history easily.

Common Immutable.js Data Structures

Immutable Structure	JavaScript Equivalent
Map	Object {}
List	Array []
Set	Set

✅Example Usage in React

npm install immutable

1. Using Map for state

```
import { Map } from 'immutable';
import React, { useState } from 'react';
function Counter() {
  const [state, setState] = useState(Map({ count: 0 }));
  const increment = () => {
    setState(prevState => prevState.update('count', val => val + 1));
  };
  return (
    <div>
      <h2>Count: {state.get('count')}</h2>
      <button onClick={increment}>Increment</button>
```

```
    </div>
  );
}
```

2. Using List for arrays

```
import { List } from 'immutable';
const [list, setList] = useState(List([1, 2, 3]));
setList(prev => prev.push(4)); // Add an element immutably
```

Service Side Rendering

Server-Side Rendering (SSR) in React refers to the process of rendering React components on the server (Node.js) rather than in the browser. The server sends the fully rendered HTML to the client, improving performance and SEO. SSR is commonly used with **Next.js**, a React framework that makes SSR easy.

✅Benefits of Server-Side Rendering (SSR)

- **Faster First Paint (especially on slow devices)**
- **Improved SEO** (important for content-heavy apps)
- **Pre-rendered HTML** sent to the client, improving crawlability

How SSR Works in React (with Next.js)

1. **Request** sent to server (e.g., https://yourdomain.com/products)
2. Server runs React code and fetches necessary data
3. Server renders the React component to HTML
4. Server sends the HTML to the browser
5. Browser hydrates the page (adds event listeners)

Example using Next.js (React SSR Framework)

Install Next.js:

UNIT- III

REACT JS

```
npx create-next-app@latest my-ssr-app
cd my-ssr-app
npm run dev
```

Create a page with SSR:

```
// pages/products.js
export async function getServerSideProps() {
  const res = await fetch('https://fakestoreapi.com/products');
  const products = await res.json();
  return { props: { products } };
}
export default function Products({ products }) {
  return (
    <div>
      <h1>Products</h1>
      {products.map(p => (
        <div key={p.id}>
          <h2>{p.title}</h2>
          <p>{p.price}</p>
        </div>
      ))}
    </div>
  );
}
```

Key Concepts in SSR

Concept	Description
getServerSideProps	Runs **on every request** to fetch data on server

UNIT- III

REACT JS

Concept	Description
getStaticProps	For **Static Site Generation** (not SSR)
Hydration	Client React takes over server-rendered HTML
SEO Optimization	Meta tags, structured data improve SEO ranking

Unit Testing

Unit testing in React means testing individual components or functions in isolation to ensure they behave as expected.

✅Popular Libraries for React Unit Testing

Tool	Purpose
Jest	Test runner + assertion library
React Testing Library (RTL)	Interact with components like a user
Enzyme	(Legacy) Shallow rendering (not recommended anymore)

➡ The recommended stack today is **Jest + React Testing Library (RTL)**.

Install Required Packages

npm install --save-dev jest @testing-library/react @testing-library/jest-dom

Add this to your package.json:

```
"scripts": {
  "test": "jest"
}
```

UNIT- III

REACT JS

Sample React Component

Counter.js

```
import React, { useState } from 'react';
export default function Counter() {
  const [count, setCount] = useState(0);
  return (
    <div>
      <h1>Count: {count}</h1>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}
```

Unit Test File

Counter.test.js

```
import React from 'react';
import { render, screen, fireEvent } from '@testing-library/react';
import Counter from './Counter';
import '@testing-library/jest-dom';

test('renders initial count', () => {
  render(<Counter />);
  expect(screen.getByText(/count: 0/i)).toBeInTheDocument();
});

test('increments count on button click', () => {
  render(<Counter />);
```

```
  fireEvent.click(screen.getByText(/increment/i));
  expect(screen.getByText(/count: 1/i)).toBeInTheDocument();
});
```

▸ Run Tests

npm test

You'll see output like:

PASS ./Counter.test.js

✓ renders initial count (xx ms)

✓ increments count on button click (xx ms)

Webpack

Webpack in React is a **module bundler** that compiles your JavaScript, CSS, images, and other assets into optimized bundles for the browser. It's commonly used in custom React setups when you're not using tools like **Create React App** or **Next.js**.

Why Use Webpack in React?

- Bundles JS, CSS, images, fonts, etc.
- Supports hot-reloading and development server
- Allows JSX, ES6+ transpiling via Babel
- Tree-shaking and optimization for production

✅Step-by-Step Setup: React + Webpack + Babel

Folder Structure

```
my-react-app/
├── public/
│   └── index.html
├── src/
```

UNIT- III

REACT JS

```
│   ├── App.js
│   └── index.js
├── webpack.config.js
├── .babelrc
├── package.json
```

1 Initialize Project

```
mkdir my-react-app && cd my-react-app
npm init -y
```

2 Install Dependencies

```
npm install react react-dom
npm install --save-dev webpack webpack-cli webpack-dev-server
npm install --save-dev babel-loader @babel/core @babel/preset-env @babel/preset-react
npm install --save-dev html-webpack-plugin
```

3 Babel Configuration (.babelrc)

```
{
  "presets": ["@babel/preset-env", "@babel/preset-react"]
}
```

4 Webpack Config (webpack.config.js)

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'main.js',
    path: path.resolve(__dirname, 'dist'),
    clean: true,
  },
  mode: 'development',
```

```
  module: {
    rules: [
      {
        test: /\.(js|jsx)$/,
        exclude: /node_modules/,
        use: 'babel-loader'
      },
      {
        test: /\.css$/i,
        use: ['style-loader', 'css-loader'],
      }
    ]
  },
  resolve: {
    extensions: ['.js', '.jsx']
  },
  plugins: [
    new HtmlWebpackPlugin({
      template: './public/index.html'
    })
  ],
  devServer: {
    static: './dist',
    hot: true,
    port: 3000
  }
};
```

5 React Files

public/index.html

UNIT- III

REACT JS

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>React App</title>
  </head>
  <body>
    <div id="root"></div>
  </body>
</html>
```

src/App.js

```
import React from 'react';

const App = () => {
  return <h1>Hello from Webpack + React</h1>;
};

export default App;
```

src/index.js

```
import React from 'react';
import { createRoot } from 'react-dom/client';
import App from './App';

const root = createRoot(document.getElementById('root'));
root.render(<App />);
```

6 Add Scripts to package.json

```
"scripts": {
  "start": "webpack serve --open",
  "build": "webpack"
}
```

▸ Run the App

```
npm start
```

Open http://localhost:3000 to see your app.