UNIT-V

React

Need of React

React is a **JavaScript library** for building **user interfaces (UIs)**, particularly **single-page applications (SPAs)**. It was developed by Facebook and is widely used in modern web development. Here's why React is needed:

1. Component-Based Architecture

React allows developers to break the UI into **reusable, self-contained components**.

- Easier to maintain and test
- Promotes code reuse and modularity
- Encourages separation of concerns

2. Efficient Updates with Virtual DOM

React uses a **Virtual DOM** to update only the parts of the page that change, rather than reloading the whole UI.

- Improves performance
- Reduces unnecessary DOM manipulations
- Enhances user experience

3. Declarative Syntax

You describe *what* the UI should look like, and React takes care of updating the DOM to match.

- Easier to read and understand
- Makes UI predictable and debuggable

4. Strong Ecosystem and Community

React has a huge ecosystem of tools, libraries, and community support.

- Rich set of third-party components
- Integration with Redux, React Router, etc.
- Large community, extensive documentation, and job demand

5. Cross-Platform Development

React powers **React Native**, which allows building **mobile apps** using the same principles and architecture.

6. SEO-Friendly (with SSR)

With **Next.js** or other SSR tools, React can be used to build SEO-friendly applications.

7. JSX – JavaScript + HTML

React uses JSX, which allows HTML to be written inside JavaScript.

- Makes code more readable
- Brings structure and logic together

Simple React Structure

1. Folder Structure

```
my-react-app/
├── public/
│   └── index.html      <-- Main HTML file
├── src/
│   ├── App.js         <-- Main App component
│   ├── index.js       <-- Entry point of the app
│   └── components/      <-- Reusable components
│       └── Hello.js
├── package.json        <-- Project config & dependencies
```

2. Core Files Explained

□ *index.html (in public/)*

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

□ *index.js (in src/)*

```
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 Hello from './components/Hello';

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

export default App;
```

□ *components/Hello.js*

```
import React from 'react';

function Hello() {
  return <p>Hello from a component!</p>;
}

export default Hello;
```

3. To Run This App

1. Install Node.js
2. Create app:

   ```
   npx create-react-app my-react-app
   ```

3. Replace the contents with your code

4. Start the app:

   ```
   npm start
   ```

The Virtual DOM

The **Virtual DOM (VDOM)** is a **lightweight in-memory copy** of the **real DOM** (Document Object Model). React uses it to optimize and efficiently update the UI.

How It Works

1. **You write JSX** (UI code in React)
2. **React builds a Virtual DOM tree** representing the current UI
3. When something changes (like a button click):
   - React **creates a new Virtual DOM**
   - It **compares it to the previous one** (using a process called **"diffing"**)
   - It **calculates the minimal set of changes**
4. **Only the necessary parts of the real DOM** are updated

Why Is Virtual DOM Fast?

- **Real DOM operations are slow** (repainting, reflowing, layout changes)
- The **Virtual DOM reduces direct interactions** with the real DOM
- React batches and optimizes updates

Example

Suppose this is your JSX:

```
<div>
  <h1>Hello</h1>
  <p>Count: 0</p>
</div>
```

If Count changes to 1:

- React **creates a new Virtual DOM** with <p>Count: 1</p>
- It **diffs** the old and new VDOMs
- It sees only the <p> tag changed
- React **updates just that part** of the real DOM, not the whole UI

Benefits of Virtual DOM

- **Better performance**
- **Faster updates**
- **Simplified programming model**
- **Efficient UI rendering**

React Components

React components are **reusable, self-contained pieces of UI**. You can think of them as custom HTML elements that you define yourself.

Types of Components

1. Functional Components ✅(Most Common)

Simpler and modern way to write components using **functions**.

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

Or using **arrow functions**:

```
const Greeting = () => <h1>Hello, World!</h1>;
```

2. Class Components □ (Older Style)

Uses ES6 class syntax. Supports **state** and **lifecycle methods** (still supported but less common with React Hooks now).

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

class Greeting extends Component {
  render() {
    return <h1>Hello from Class!</h1>;
  }
}
```

Reusability Example

You can reuse a component like this:

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

Props — Passing Data to Components

Props are like **function arguments** that allow you to **pass data into components**.

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

Usage:

```
<Greeting name="Alice" />
<Greeting name="Bob" />
```

Stateful Components (with Hooks)

Use useState to add **state** in functional components:

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

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

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

Introducing React Components

React components are at the **heart of React development**. They allow you to **build complex user interfaces** by breaking them into smaller, **reusable pieces**.

A **React component** is a **function or class** that returns a **piece of UI**, typically written in **JSX** (JavaScript + HTML-like syntax).

Think of components like **custom HTML tags** that you create.

Why Use Components?

- **Reusable** – Write once, use many times
- **Organized** – Separate logic and UI into small, manageable pieces
- **Dynamic** – Accept **props** to customize behavior and appearance

Example 1: Functional Component (Modern Way)

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

Usage:

```
<Welcome />
```

Example 2: Component with Props

```
function Greeting(props) {
  return <h2>Hello, {props.name}!</h2>;
}
```

Usage:

```
<Greeting name="Alice" />
<Greeting name="Bob" />
```

Example 3: Component with State (using Hooks)

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

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

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

How to Use Components in an App

```
import React from 'react';
import Greeting from './Greeting';

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

export default App;
```

Creating Components in React

In React, **components** are the foundation of any app. Let's go step-by-step through how to **create components** and use them.

Step 1: Functional Component (Most Common)

Here’s a basic example of creating and using a **functional component**:

```
// Hello.js
import React from 'react';

function Hello() {
  return <h1>Hello from React!</h1>;
}
```

```
export default Hello;
```

Usage in App.js:

```
import React from 'react';
import Hello from './Hello';

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

export default App;
```

Step 2: Component with Props (Passing Data)

```
// Greet.js
import React from 'react';

function Greet(props) {
  return <h2>Hello, {props.name}!</h2>;
}

export default Greet;
```

Usage:

```
<Greet name="Alice" />
<Greet name="Bob" />
```

Step 3: Component with State (Using Hooks)

```
// Counter.js
import React, { useState } from 'react';

function Counter() {
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click Me</button>
    </div>
  );
}

export default Counter;
```

Usage in App.js:

```
<Counter />
```

Step 4: Organize Your Files

Typical structure:

```
├── components/
│   ├── Hello.js
│   ├── Greet.js
│   └── Counter.js
├── App.js
└── index.js
```

Step 5: Render in index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

ReactDOM.render(<App />, document.getElementById('root'));
```

Data and Data Flow in React

In React, **data flows in one direction — from parent to child**. Understanding how data flows between components is key to building well-structured, maintainable apps.

□ Types of Data in React

Data Type	**Description**

Data Type	**Description**
Props	Read-only data passed from parent to child
State	Local, mutable data managed within a component

1. Props (Short for "Properties")

- Used to **pass data from parent to child**
- **Read-only** (a child component cannot change props)

Example:

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

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

2. State

- State is **data owned and controlled** by a component
- State can **change over time**, triggering a **re-render**

Example using useState:

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

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

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

Data Flow Diagram

Parent (state or props)
↓
Child (props only)

- **Parent → Child**: Use props
- **Child → Parent**: Use a callback function (e.g., send data back via props)

3. Passing Data Back to Parent

You can pass a **function from parent to child** and call it in the child to send data back.

Example:

```
function Child({ sendData }) {
  return <button onClick={() => sendData('Hello from Child')}>Click</button>;
}

function Parent() {
  const handleData = (msg) => {
    alert(msg);
  };

  return <Child sendData={handleData} />;
}
```

4. State Management for Larger Apps

- For small apps: useState, useContext
- For large apps: Tools like **Redux**, **Zustand**, or **React Query**

Rendering and Life Cycle Methods in React

Understanding how **rendering** and **lifecycle methods** work in React is essential for building **efficient and responsive UIs**.

1. Rendering in React

Rendering is the process of **displaying components** on the screen. React does this using:

- **JSX** to describe what the UI should look like

- A **Virtual DOM** to efficiently update the real DOM

Example:

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

React **renders** this when you use:

```
<Welcome />
```

2. Lifecycle of a Component

Each component in React has a **lifecycle** — phases from **creation to removal**.

There are **three main phases**:

Phase	Description
Mounting	Component is created and inserted into the DOM
Updating	Component re-renders due to state/prop change
Unmounting	Component is removed from the DOM

3. Lifecycle Methods (Class Components)

Note: Lifecycle methods are available in **class components**. In **functional components**, we use **Hooks** (like useEffect) instead.

Mounting (Creation)

```
componentDidMount()
```

- Called **once** after the component is rendered
- Good for **API calls** or setting up subscriptions

Updating (Re-rendering)

componentDidUpdate(prevProps, prevState)

- Called when **state or props change**
- Useful for responding to updates

Unmounting (Removal)

componentWillUnmount()

- Called before the component is removed
- Good for **clean-up tasks** (e.g., clearing timers, unsubscribing)

Example (Class Component)

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

class Timer extends Component {
  componentDidMount() {
    console.log('Component mounted');
  }

  componentDidUpdate() {
    console.log('Component updated');
  }

  componentWillUnmount() {
    console.log('Component will unmount');
  }

  render() {
    return <h1>Timer Component</h1>;
  }
}
```

Lifecycle with Hooks (Functional Components)

In **functional components**, we use the useEffect Hook to handle lifecycle logic.

useEffect for All Phases:

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

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

  useEffect(() => {
    console.log('Component mounted or updated');

    return () => {
      console.log('Component will unmount');
    };
  }, [count]); // runs when 'count' changes

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

Working with forms in React

Forms in React work a little differently than in plain HTML. React uses **controlled components** to manage form data using **state**.

1. Basic Controlled Form Example

A **controlled component** is a form input element whose value is **controlled by React state**.

Example: Simple Input Form

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

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

  const handleSubmit = (event) => {
    event.preventDefault(); // Prevent page reload
    alert(`Submitted Name: ${name}`);
  };
  return (
    <form onSubmit={handleSubmit}>
      <label>
        Name:
        <input
          type="text"
          value={name}
          onChange={(e) => setName(e.target.value)}
        />
      </label>
      <button type="submit">Submit</button>
    </form>
  );
}

export default NameForm;
```

2. Multiple Inputs with One Handler

```
function ContactForm() {
  const [formData, setFormData] = useState({
    name: ",
    email: "
  });

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

  const handleSubmit = (e) => {
    e.preventDefault();
    console.log(formData);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input name="name" value={formData.name} onChange={handleChange} placeholder="Name" />
      <input name="email" value={formData.email} onChange={handleChange} placeholder="Email" />
      <button type="submit">Send</button>
    </form>
  );
}
```

3. Controlled vs Uncontrolled Components

Feature	Controlled Component	Uncontrolled Component
Data source	React state	DOM (ref)
Real-time validation	Easy	Harder
Flexibility	More flexible and predictable	Less flexible

4. Handling Other Inputs

Checkbox

```
<input
  type="checkbox"
  checked={isChecked}
  onChange={(e) => setIsChecked(e.target.checked)}
/>
```

Select

```
<select value={fruit} onChange={(e) => setFruit(e.target.value)}>
  <option value="apple">Apple</option>
  <option value="banana">Banana</option>
</select>
```

Integrating third party libraries

React lets you easily use **third-party libraries** to add extra functionality, UI components, or utilities to your app.

How to Integrate Third-Party Libraries

Step 1: Install the library

Use **npm** or **yarn**:

```
npm install library-name
# or
yarn add library-name
```

Step 2: Import and Use in Your Component

```
import LibraryComponent from 'library-name';

function MyComponent() {
  return <LibraryComponent />;
}
```

Common Use Cases

1. UI Component Libraries

- **Material-UI (MUI)**
- **Ant Design**
- **Bootstrap React**

Example:

```
npm install @mui/material @emotion/react @emotion/styled
```

```
import Button from '@mui/material/Button';

function App() {
  return <Button variant="contained">Click Me</Button>;
}
```

2. Utility Libraries

- **Lodash** (helpers)
- **Axios** (HTTP requests)
- **date-fns / Moment.js** (date formatting)

Example with Axios:

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

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

  useEffect(() => {
    axios.get('https://api.example.com/items')
      .then(res => setData(res.data))
      .catch(err => console.error(err));
  }, []);

  return <div>{data ? JSON.stringify(data) : 'Loading...'}</div>;
}
```

3. React-Specific Libraries

Some libraries are built for React or have React wrappers, e.g.,

- **React Router** (routing)
- **React Query** (data fetching)
- **Formik / React Hook Form** (form management)

Tips for Integration

- **Check compatibility** with your React version
- Use **ES Modules imports** when available
- Follow the library’s **documentation** for React-specific usage
- Use **React wrappers/components** for better integration

Routing in React

Routing allows you to **navigate between different pages or views** in a React app without refreshing the browser. The most popular routing library in React is **React Router**.

How to Add Routing with React Router

Step 1: Install React Router

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

Step 2: Set Up Routes in Your App

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

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

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

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

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

export default App;
```

Key Concepts

Concept	**Description**
<Router>	Wraps your app and enables routing
<Routes>	Container for all your route definitions
<Route>	Defines a route: path + component to render
<Link>	Navigation links without page reload

Navigating Programmatically

Use the useNavigate hook:

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

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

  const handleLogin = () => {
    // After login success
    navigate('/dashboard');
  };

  return <button onClick={handleLogin}>Login</button>;
}
```

Nested Routes Example

```
function Dashboard() {
  return (
    <div>
      <h2>Dashboard</h2>
      <Routes>
        <Route path="profile" element={<Profile />} />
        <Route path="settings" element={<Settings />} />
      </Routes>
    </div>
  );
}
```