React.js

# 1. React Components

### Definition:

Components are the **building blocks** of a React application. They allow developers to break the UI into **independent**, **reusable**, and **manageable** pieces, each handling its own logic and rendering.

### Types of Components:

1. **Functional Components**

These are **plain JavaScript functions** that return JSX (JavaScript XML).

Simple and preferred for most use-cases in modern React (especially with hooks).

**Example:**

function Welcome() {

return <h1>Hello, React!</h1>;

}

1. **Class Components**

These are based on **ES6 classes** and extend from React.Component.

Used for managing **state** and lifecycle methods (before hooks became standard).

**Example:**

class Welcome extends React.Component {

render() {

return <h1>Hello, React!</h1>;

}

}

### Advantages of Using Components:

* **Reusability:**  
  Build once, use multiple times across the app.
* **Separation of Concerns:**  
  Each component handles its own logic, making code more organized and modular.
* **Ease of Testing:**  
  Smaller components are easier to isolate and test individually.
* **Improved Readability & Maintainability:**  
  Cleaner structure and better collaboration among teams.

### Disadvantages:

* **Over-Splitting:**  
  Creating too many tiny components can lead to a **complex and deeply nested structure**, making the app harder to understand and debug.

# 2. Props (Properties)

### Definition:

Props are **inputs to React components**. They are used to pass **data from parent to child components** and are **read-only**, meaning the child cannot modify them.

Props help make components **dynamic and customizable** based on the data they receive.

### Example:

function Greet(props) {

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

}

// Usage

<Greet name="John" />

* In this example, the name prop is passed to the Greet component.
* The component renders: **Hello, John!**

### Advantage:

* **Allows Dynamic Content:**  
  Components can render different outputs based on the props they receive, making them more flexible and reusable.

### Disadvantage:

* **Read-Only Nature:**  
  Props **cannot be modified** inside the child component, which can be limiting if you want to update data within the child.

# 3. State

### Definition:

**State** is a **built-in object** used to **store dynamic data** in a component.  
It represents information that can **change over time** based on user interactions, network responses, or any internal logic.

In functional components, state is managed using the **useState hook**.

### Example using useState:

import { useState } from 'react';

function Counter() {

const [count, setCount] = useState(0); // count is the state variable, setCount updates it

return (

<div>

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

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

</div>

);

}

* When the button is clicked, count is incremented and the UI updates automatically.

### Advantage:

* **Enables Dynamic Interactivity:**  
  Allows components to **respond to user input** or other events in real-time by updating the UI based on state changes.

### Disadvantage:

* **Performance Overhead:**  
  If not managed properly, frequent or unnecessary state updates can **impact performance**, especially in large applications.

# 4. Hooks

### Definition:

**Hooks** are **special functions** in React that let you **use state, lifecycle methods, and other React features** inside **functional components**, which were traditionally limited to rendering UI only.

Introduced in **React 16.8**, hooks made functional components more powerful and concise.

### Commonly Used Hooks:

1. **useState()** – Manages state in a functional component.
2. **useEffect()** – Handles side effects like data fetching, subscriptions, or manually changing the DOM.
3. **useRef()** – Accesses and persists mutable values without causing re-renders.
4. **useContext()** – Accesses the value from a context provider.
5. (Others include: useMemo(), useCallback(), useReducer() etc.)

### Example using useState() and useEffect():

import { useEffect, useState } from 'react';

function Example() {

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

useEffect(() => {

fetch('https://api.example.com')

.then(res => res.json())

.then(data => setData(data));

}, []); // empty dependency array = run only once after initial render

return <div>{JSON.stringify(data)}</div>;

}

* useState manages the data.
* useEffect fetches API data once when the component mounts.

### Advantage:

* **Simplifies Functional Components:**  
  Hooks remove the need for class components, making code cleaner, shorter, and easier to manage.

### Disadvantage:

* **Complexity in Large Components:**  
  Using many hooks in a single component can lead to **unorganized logic**, making it harder to debug and maintain.

# 5. Lifecycle Methods

### In Class Components:

React provides specific **lifecycle methods** to control different phases of a component’s existence:

* componentDidMount() – Runs **once** after the component is mounted.
* componentDidUpdate() – Runs **after every update**.
* componentWillUnmount() – Runs **before the component is unmounted**.

### Functional Alternative: useEffect()

In functional components, these lifecycle behaviors are handled using the versatile **useEffect()** hook.

### Example: Mounting & Unmounting in Functional Components

import { useEffect } from 'react';

useEffect(() => {

console.log("Component Mounted");

return () => {

console.log("Component Unmounted");

};

}, []); // empty dependency array mimics componentDidMount + componentWillUnmount

* **Mount:** The function runs once when the component loads.
* **Unmount:** The return function runs when the component is removed.

### Advantage:

* **Helps Manage Setup and Cleanup:**  
  With useEffect(), you can easily **set up listeners, fetch data, or start timers**—and **clean them up** when needed.

### Disadvantage:

* **Improper Use May Cause Bugs:**  
  Misplacing dependencies or nesting useEffect() incorrectly can lead to **infinite loops, stale values, or memory leaks**.

# 6. Forms

### Definition:

React handles form elements through **controlled components**, where the **form data is managed by React state**.  
This ensures the form inputs behave predictably and stay in sync with the UI.

### Example – Controlled Form Input:

import { useState } from 'react';

function FormExample() {

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

const handleSubmit = (e) => {

e.preventDefault(); // Prevents page reload

alert(`Name: ${name}`);

};

return (

<form onSubmit={handleSubmit}>

<input

type="text"

value={name}

onChange={e => setName(e.target.value)}

/>

<button type="submit">Submit</button>

</form>

);

}

* value={name} makes the input a **controlled component**.
* onChange updates the state in real-time as the user types.

### Advantage:

* **Controlled Inputs Ensure Predictable Data:**  
  React state reflects the current input values at all times, making validation, formatting, and debugging easier.

### Disadvantage:

* **More Code Compared to Plain HTML Forms:**  
  You need to write handlers and state logic for each input, which adds **extra boilerplate** for large forms.

# 7. Event Handling

### Definition:

React handles events using **camelCase syntax** (e.g., onClick, onChange) and passes a **function reference** as the event handler.

This approach is similar to handling events in plain JavaScript, but with JSX syntax and consistent binding.

### Example: Handling a Button Click

function ClickButton() {

const handleClick = () => alert("Button Clicked");

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

}

* onClick is the event name (camelCase).
* handleClick is the handler function that runs when the button is clicked.

### Advantage:

* **Easy to Bind Handlers in Functional Components:**  
  No need for manual binding; functions can be directly passed as handlers.

### Disadvantage:

* **Binding in Class Components Can Be Tricky:**  
  In class components, you often need to bind this manually in the constructor or use arrow functions to avoid scope issues.

# 8. Conditional Rendering

### Definition:

Conditional rendering means **displaying different UI elements** based on a condition or state.  
It allows components to **dynamically adapt** to different situations or user actions.

### Example: Using Ternary Operator

{isLoggedIn ? <Dashboard /> : <Login />}

* If isLoggedIn is true, it renders the <Dashboard />.
* If false, it shows the <Login />.

### Advantage:

* **Flexible UI:**  
  Easily adjust what is displayed based on user interaction, state, or props.

### Disadvantage:

* **Reduced Readability with Nested Conditions:**  
  Using multiple or nested ternary operators can make the JSX messy and **hard to read or maintain**.

# 9. Routing

### Setup:

To use React Router in your project, you first need to install it:

npm install react-router-dom

### Example: Basic Routing Setup

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

function App() {

return (

<BrowserRouter>

<Routes>

<Route path="/" element={<Home />} />

<Route path="/about" element={<About />} />

</Routes>

</BrowserRouter>

);

}

* BrowserRouter wraps the entire app to handle routing.
* Routes contains all your Route elements.
* Each Route specifies a path and the element to render when that path is visited.

### Advantage:

* **Single Page App (SPA) Experience:**  
  React Router enables **navigation without reloading the page**, creating a fluid, faster, and more seamless user experience typical of SPAs.

### Disadvantage:

* **SEO Limitations (unless SSR is used):**  
  Since React Router relies on client-side rendering, **search engines** may not index content properly unless you use **Server-Side Rendering (SSR)** or other techniques like **React Helmet** for managing meta tags.

# 10. Storage (Local, Session, Cookies)

### LocalStorage:

* **Persistence:** Data stored in LocalStorage is **persistent** and remains until manually cleared (e.g., through the browser settings).
* **Example:**
* localStorage.setItem('user', 'John');
* console.log(localStorage.getItem('user')); // Output: 'John'
* **Use Case:** Storing user preferences or other non-sensitive data.

### SessionStorage:

* **Persistence:** Data stored in SessionStorage is **only available for the duration of the page session** (i.e., as long as the tab or window is open). It is cleared when the tab is closed.
* **Example:**
* sessionStorage.setItem('token', 'abc');
* console.log(sessionStorage.getItem('token')); // Output: 'abc'
* **Use Case:** Storing temporary data that doesn’t need to persist between sessions (e.g., login tokens during a single session).

### Cookies:

* **Persistence:** Cookies have an **expiration date** defined by the expires attribute. If not specified, they are treated as session cookies and are deleted when the browser is closed.
* **Example:**
* document.cookie = "user=John; expires=Fri, 31 Dec 2025 12:00:00 UTC";
* **Use Case:** Storing user authentication tokens, preferences, or other small pieces of data that need to persist across sessions or be sent to the server.

### Advantage:

* **Stores User Data on Client-Side:**  
  These storage options allow storing data locally, improving performance by reducing server calls and maintaining state across sessions.

### Disadvantage:

* **Can Be Accessed and Modified:**  
  Since these storage options are **client-side**, they can be accessed and potentially modified by users or malicious scripts. Therefore, they should **not be used for storing sensitive data** (e.g., passwords, personal info).

# 11. Fetching APIs (using fetch and axios)

## Fetching APIs in React with useEffect and fetch

### Using fetch with useEffect:

The fetch function is used to **make HTTP requests** in JavaScript. In React, useEffect is commonly used to trigger data fetching when a component mounts.

import { useEffect, useState } from 'react';

function FetchData() {

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

useEffect(() => {

async function fetchData() {

const res = await fetch('https://api.example.com/data');

const json = await res.json();

setData(json);

}

fetchData();

}, []); // Empty dependency array ensures fetch happens only once

return <div>{JSON.stringify(data)}</div>;

}

* **fetchData()** is an asynchronous function that fetches data from the API.
* **setData(json)** updates the state with the fetched data.

### Advantage:

* **Connects Frontend with Backend/API:**  
  fetch provides a simple way to interact with APIs, allowing you to **send requests** to the backend and **render responses** in your component.

### Disadvantage:

* **Needs Error and Async Handling:**  
  fetch does not handle errors automatically, so you need to **manually manage error cases** (e.g., network failures, non-2xx status codes) and ensure **asynchronous operations** are handled properly.

Example with basic error handling:

useEffect(() => {

async function fetchData() {

try {

const res = await fetch('https://api.example.com/data');

if (!res.ok) throw new Error('Network response was not ok');

const json = await res.json();

setData(json);

} catch (error) {

console.error('Fetching error:', error);

}

}

fetchData();

}, []);

## Fetching APIs in React with axios

### Setup Axios:

First, install Axios in your project:

npm install axios

### Using axios with useEffect:

axios is a promise-based HTTP client that makes fetching APIs cleaner and provides built-in error handling.

import { useEffect, useState } from 'react';

import axios from 'axios';

function FetchData() {

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

const [error, setError] = useState(null);

useEffect(() => {

async function fetchData() {

try {

const response = await axios.get('https://api.example.com/data');

setData(response.data);

} catch (error) {

setError('Error fetching data');

console.error('Fetching error:', error);

}

}

fetchData();

}, []); // Empty dependency array ensures fetch happens only once

if (error) {

return <div>{error}</div>;

}

return <div>{JSON.stringify(data)}</div>;

}

* **axios.get()** sends a GET request to the API.
* **response.data** contains the data returned from the API.

### Advantage:

* **Simpler Error Handling and Configuration:**  
  Axios automatically handles errors, and its syntax is cleaner compared to fetch, making it easier to manage responses and handle failures.
* **Supports Request Cancellation:**  
  Axios provides built-in support for canceling requests, which can be useful when working with multiple API calls.

### Disadvantage:

* **Additional Dependency:**  
  You have to install axios as an external package, whereas fetch is built into the browser.