React Js – FrontEnd Developer

**1. Introduction to React JS**

**💡 What is React JS?**

**React JS** is a **JavaScript library** created by **Facebook** for building **user interfaces**, especially for **web applications**. It helps developers build **fast, interactive**, and **dynamic** websites or apps.

* 📦 React is component-based – your UI is built using reusable "components" (small blocks).
* ⚡ It updates the web page **efficiently** using a concept called the **Virtual DOM**.

**🧠 Why use React?**

* Easier to manage large apps
* Reusable code (components)
* Great performance
* Huge community & support

**✅ 2. What is a Single Page Application (SPA)?**

**🧭 Traditional Website:**

* Each time you click a link, the browser **loads a new HTML page** from the server.
* It’s slower and uses more bandwidth.

**🏎️ SPA (Single Page Application):**

* Loads **one HTML page** at the start.
* After that, it only updates **parts of the page** when needed (using JavaScript).
* Navigation feels fast and smooth – **no full page reloads**!

✅ React is perfect for SPAs – it updates just what’s needed.

**✅ 3. What is the Virtual DOM?**

**🧱 What is the DOM?**

* The **DOM (Document Object Model)** is the structure of the webpage in memory (like a tree of elements).

**🧠 Virtual DOM (in React):**

* A **copy of the real DOM** kept in memory by React.
* When something changes, React:
  1. Updates the **Virtual DOM**.
  2. **Compares** it with the previous version (diffing).
  3. Finds **what changed**.
  4. Only updates **that part** of the real DOM.

🔥 This makes React apps much **faster** and more **efficient**.

**✅ 4. What is JSX?**

**🖋️ JSX = JavaScript + HTML**

* JSX is a **syntax extension** for JavaScript that lets you **write HTML inside JavaScript**.

Example:

**jsx**

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

Instead of writing:

**js**

**document.createElement("h1")**

✅ JSX is easier to write and understand  
✅ JSX looks like HTML, but it gets **converted to JavaScript** behind the scenes

**🔄 Summary**

| **Term** | **Explanation** |
| --- | --- |
| **React JS** | A JavaScript library for building UI using components |
| **SPA** | Single-page app that loads once and updates content dynamically |
| **Virtual DOM** | A fast, in-memory copy of the DOM used to update only changed parts |
| **JSX** | A syntax to write HTML-like code inside JavaScript |

**Day - 2**

**1️⃣ What is a “React app” vs a “Vite app”?**

* When people say **“React app”**, they often mean *React created with Create React App (CRA)*.
* **Vite** is a *build tool* that can also create React apps, but it’s faster and more modern than CRA.

| **Feature** | **Create React App (CRA)** | **Vite (with React)** |
| --- | --- | --- |
| **Purpose** | Official tool (from React team) to set up a React project quickly. | Modern build tool to create web apps (including React, Vue, Svelte, etc.). |
| **Bundler** | Uses **Webpack** (slower builds) | Uses **ESBuild** (written in Go, extremely fast) for dev + Rollup for production |
| **Startup speed** | Slower to start dev server (especially in big apps) | Lightning-fast dev server startup |
| **Hot reload speed** | Can be sluggish in big projects | Very fast HMR (Hot Module Replacement) |
| **Config** | Minimal out-of-the-box, but hard to customize without eject | Easy to configure with plugins and options |
| **Popularity trend** | Declining (CRA is considered outdated) | Rapidly growing (favored in 2024–2025) |

**2️⃣ Why is everybody using Vite instead of CRA now?**

**Main reasons:**

1. **⚡ Speed** – Vite is *way* faster in:
   * Starting the dev server
   * Applying code changes
   * Building the app for production  
     *(CRA’s Webpack build is slow, especially on big projects.)*
2. **🛠 Easier customization** – With CRA, you must eject to tweak Webpack configs (which is messy).  
   Vite has an easy vite.config.js file for customization.
3. **📦 Smaller, cleaner output** – Production builds are smaller and optimized.
4. **📈 Modern tech stack** –
   * ESBuild (fast compilation)
   * Rollup (optimized bundling)
   * Out-of-the-box support for TypeScript, JSX, CSS modules, etc.
5. **🚀 Active development** – CRA’s updates are slow, but Vite is actively maintained with modern features.

✅ **In short:**

* CRA = older, slower, React-only
* Vite = faster, modern, supports multiple frameworks (including React)

**Day-3**

**1️⃣ What is a Functional Component in React?**

Think of a **component** like a **small reusable block** of your website.  
Instead of writing one giant HTML file, you split it into **pieces** (components) and combine them.

A **functional component** is just a **JavaScript function** that returns JSX (HTML-like code).

💡 **Analogy:**  
If your website is a *Lego house*, each Lego brick is a **component**.

**Example of a Functional Component**

function Welcome() {

return <h1>Hello, welcome to my website!</h1>;

}

export default Welcome;

Here:

* Welcome is a **function** (component name must start with a **capital letter**).
* It returns JSX (HTML inside JavaScript).
* You can use <Welcome /> anywhere in your app.

**2️⃣ What are Props?**

**Props** (short for “properties”) are like **inputs** you pass into a component to change how it looks or behaves.

💡 **Analogy:**  
Think of a **coffee machine** (component).  
You can pass different **coffee beans** (props) to get different coffee flavors.

**Example with Props**

function Welcome(props) {

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

}

export default Welcome;

And in another file:

import Welcome from './Welcome';

function App() {

return (

<div>

<Welcome name="Alice" />

<Welcome name="Bob" />

</div>

);

}

export default App;

📝 **What’s happening here?**

* <Welcome name="Alice" /> → sends the prop name with value "Alice".
* Inside the Welcome component, props.name becomes "Alice".
* This makes the output:

Hello, Alice!

Hello, Bob!

**3️⃣ Key Points to Remember**

* Functional components = **functions** that return JSX.
* Props = **data** you send into components.
* Props are **read-only** (you can’t change them inside the component).
* Components make your code **reusable** and **easier to manage**.

**Day – 4**

**Why Style in React?**

Styling makes your app look nice — it controls colors, fonts, layout, and animations. React gives you several ways to apply styles.

Today, we’ll focus on:

* CSS Modules
* Styled-Components

**1. CSS Modules**

**🔹 What is it?**

**CSS Modules** are just regular .css files, but scoped to a single component. This means **no style conflicts** — perfect when you're building component-based UIs.

**🔧 How to Use CSS Modules**

1. Create a .module.css file
2. Import it into your component
3. Use it like an object

.btn {

background-color: blue;

color: white;

padding: 10px;

border-radius: 5px;

}

**✅ Pros of CSS Modules:**

* ✅ No global style conflicts
* ✅ Easy to use if you know regular CSS
* ✅ Great for small to medium apps

**🌟 2. Styled-Components (CSS-in-JS)**

**🔹 What is it?**

Styled-Components lets you **write CSS inside your JavaScript** files. You create **styled versions of HTML elements** using a library.

You need to install it first:

npm install styled-components

**🧱 Example:**

import styled from 'styled-components';

const StyledButton = styled.button`

background-color: green;

color: white;

padding: 10px;

border-radius: 5px; ` ;

function App() {

return <StyledButton>Click Me</StyledButton>;

}

**✅ Pros of Styled-Components:**

* ✅ Styles live next to your components
* ✅ No class name bugs — it's all JavaScript
* ✅ Dynamic styling with props (very powerful)

**🆚 CSS Modules vs. Styled-Components**

| **Feature** | **CSS Modules** | **Styled-Components** |
| --- | --- | --- |
| Syntax | Traditional CSS | CSS inside JS (template literals) |
| Scope | Local by default | Local by default |
| Setup | Easy, no extra libraries | Needs installation (styled-components) |
| Dynamic Styles | Harder to do | Very easy using props |
| Performance | Very good | Slightly heavier (runtime) |

**🧠 Which One Should You Use?**

| **If you like…** | **Use…** |
| --- | --- |
| Writing CSS in .css files | **CSS Modules** |
| Keeping all logic + styles in one place | **Styled-Components** |
| Avoiding third-party libraries | **CSS Modules** |
| Building highly dynamic styles | **Styled-Components** |

**Day - 5**

**Get started with Tailwind CSS with Vite APP**  
01 - **Create your project**

* npm create vite@latest my-project
* cd my-project

02 - **Install Tailwind CSS**

* npm install tailwindcss @tailwindcss/vite

03 - **Configure the Vite plugin**

* import { defineConfig } from 'vite'

import tailwindcss from '@tailwindcss/vite'

export default defineConfig({

plugins: [

tailwindcss(),

],

})

**04 - Import Tailwind CSS**

* @import "tailwindcss";

For More Styling Informations

Visit here - <https://tailwindcss.com/docs>

**Day – 6**

🎯 What is useState?

In **React**, components can **change over time** — for example:

* A button gets clicked
* A user types into an input
* Data is fetched from an API

To **remember and manage those changes**, we use **state**.

React provides a built-in function called useState to **add state to functional components**.

**🧠 What is "state"?**

useState is a Hook that lets your component store and update data — like a variable that React watches.useState is a Hook that lets your component store and update data — like a variable that React watches.

**📦 Think of useState as:**

A way to store and update values in your component.

**🧠 Basic Syntax**

const [value, setValue] = useState(initialValue);

* value: The current state value.
* setValue: A function to update the state.
* initialValue: The starting value of the state.

**✅ Example: Counter**

import React, { useState } from 'react';

function Counter() {

const [count, setCount] = useState(0); // 👈 count starts at 0

return (

<div>

<p>You clicked {count} times</p>

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

Click Me!

</button>

</div>

);

}

**When you click the button:**

* setCount(count + 1) updates the value of count
* React **re-renders** the component
* You see the new number on the screen!

**Why use useState?**

* To **remember values** between re-renders
* To **update the UI** when something changes
* To avoid using messy class components

**Event Handling**

React event handling is a system for managing user interactions (like clicks, key presses, mouse movements, form submissions, etc.) in React components.

function handleClick(event) {

console.log(event); // SyntheticEvent

}

**2. Event Naming**

In React, event names use **camelCase** instead of lowercase (unlike HTML).

| **HTML** | **React** |
| --- | --- |
| onclick | onClick |
| onchange | onChange |
| onmouseover | onMouseOver |

**🔁 Common Events in React**

| **Event Type** | **Handler** |
| --- | --- |
| Click | onClick |
| Change | onChange |
| Submit | onSubmit |
| Focus | onFocus |
| Blur | onBlur |
| Key Down/Up | onKeyDown, onKeyUp |
| Mouse Over | onMouseOver |

import React, { useState } from "react";

function App() {

  const [inputValue, setInputValue] = useState('');

  function handleInputChange(e) {

    setInputValue(e.target.value);

  }

  return (

    <div>

      <h1>Below write sentence: {inputValue}</h1>

      <input

        type="text"

        value={inputValue}

        onChange={handleInputChange}

        placeholder="Type something here"

      />

    </div>

  );

}

export default App;

**lists & keys in react**

**📋 Lists in React**

To render a **list of items**, you typically use JavaScript's .map() method inside JSX.

function App() {

const fruits = ['Apple', 'Banana', 'Cherry'];

return (

<ul>

{fruits.map((fruit, index) => (

<li key={index}>{fruit}</li>

))}

</ul>

);

}

Keys help React **identify which items changed, were added, or removed**. This makes re-rendering more efficient.

function App() {

const users = [

{ id: 1, name: 'Alice' },

{ id: 2, name: 'Bob' },

{ id: 3, name: 'Charlie' },

];

return (

<ul>

{users.map((user) => (

<li key={user.id}>{user.name}</li>

))}

</ul>

);

}

**Global state - context api**

**🌐 What is Global State?**

Global state is data needed by multiple parts of your app. Examples:

* Logged-in user info
* Theme settings (light/dark)
* Language preference
* Cart contents (e-commerce)

**🧰 What is the Context API?**

The **Context API** lets you:

1. Create a "context" (a container for shared data)
2. Provide it (make it available to components)
3. Consume it (read or update the data from any component)

**Lifting state, state sharing patterns**

**🚀 1. Lifting State Up**

**“Lifting state up” means moving shared state to the nearest common ancestor of components that need to read or change it.**

**🔍 Why Lift State?**

**When two or more components need to share and sync data, it's best to:**

* **Store that state in their common parent**
* **Pass it down as props**
* **Update it using callback functions**

**2. State Sharing Patterns in React**

Once you outgrow lifting state (especially in large apps), these are the **state sharing patterns** you can use:

**🔹 a) Props and Lifting State Up**

* Best for **simple** or **local component trees**
* Quick and easy

🔸 **Downsides:** Gets messy with many layers (prop drilling)

**🔹 b) Context API**

* Use for **global state** (e.g., user, theme, language)
* Avoids prop drilling
* Combine with useReducer for more structure

**🔹 c) State Management Libraries**

Use for complex or large-scale apps:

* **Redux** – centralized, powerful, verbose
* **Zustand** – simpler, minimal boilerplate
* **Recoil**, **Jotai**, **MobX**, etc.

**----------------------------------------------------------------------------------------------------------------------------------------------**

**Task(22-08-2025)**

1. Create a navbar with 5 menu items.

2. Below the navbar, add an input box and a submit button.

3. Below that, display the submitted values in a list using JavaScript array methods. Each item should have two buttons: "Complete" and "Delete".

4. When a task is marked as completed, it should be visually struck through using HTML's <s> or <del> tag (or by applying a CSS class).  
When the "Delete" button is clicked, the task should be removed from the list using JavaScript methods like pop() or array filtering.

5. Style the entire layout using Tailwind CSS or CSS Modules.

(note here use – useState,event handling, lists and keys in react)

**🧠 What is useEffect?**

In React, useEffect is a **hook** that automatically renders your components whenever state or props change:

* **First shows up** on the screen (mounts)
* **Updates** (e.g. props or state change)
* **Goes away** (unmounts)

**🏗️ Why do we need useEffect?**

Some things you **can’t do directly** inside your JSX or regular component code. For example:

* Fetching data from an API
* Setting up event listeners (like scrolling or keypress)
* Running code **only once** when the component loads

That’s where useEffect comes in.

1.useEffect(() => {

console.log("Component loaded!");

}); //---🡪 render all time (every actions like button click…)

2. useEffect(() => {

console.log("Component loaded!");

}, []); // -----🡪Empty array = run once

3. useEffect(() => {

console.log("Count changed:", count);

}, [count]); //--🡪only works only given(count) component

Example

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

const Effect = () => {

    const [add, setAdd] = useState(0)

    function count(){

        setAdd(add + 1)

    }

    useEffect(()=>{

        console.log("hello")

    })

  return (

    <div>

        <h1>num = {add}</h1>

        <button onClick={count}>submit</button>

    </div>

  )

}

export default Effect

useRef

useRef is a Hook in React that lets you persist a **mutable reference** to a value across renders **without causing a re-render** when the value changes.

**🔹 Basic Syntax**

const myRef = useRef(initialValue);

**🔸 Main Use Cases**

1. **Accessing DOM elements**
2. **Storing mutable values (e.g., timers, intervals)**
3. **Tracking previous state or props**

**1. DOM Access**

import { useRef, useEffect } from 'react';

function InputFocus() {

const inputRef = useRef(null);

useEffect(() => {

inputRef.current.focus(); // focuses the input on mount

}, []);

return <input ref={inputRef} />;

}

**🧠 2. Mutable Values (Without Triggering Renders)**

function Timer() {

const count = useRef(0);

function handleClick() {

count.current += 1;

console.log("Clicked:", count.current);

}

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

}

**🧠 3. Storing Previous Values**

function PreviousValue({ value }) {

const prevValue = useRef();

useEffect(() => {

prevValue.current = value;

}, [value]);

return (

<div>

<p>Current: {value}</p>

<p>Previous: {prevValue.current}</p>

</div>

);

}

* This is useful for comparison logic or debugging

USEMEMO

useMemo is a **React Hook** that helps you **optimize performance** by **memoizing** the result of a calculation—**recomputing it only when its dependencies change**.

**🔸 Why Use useMemo?**

React re-renders components frequently. If your component does **heavy calculations** or returns **the same object/array reference** each time (causing unnecessary renders or effects), useMemo helps:

* Avoid slow computations on every render
* Prevent unnecessary re-renders in child components (due to object/array identity changes)
* **🧠 Example: Avoid Expensive Calculation**
* function ExpensiveComponent({ number }) {
* const expensiveResult = useMemo(() => {
* console.log('Calculating...');
* let total = 0;
* for (let i = 0; i < 1e9; i++) total += i;
* return total + number;
* }, [number]);
* return <div>Result: {expensiveResult}</div>;
* }
* **✅ Summary**

| **Feature** | **useMemo** |
| --- | --- |
| Purpose | Memoize a computed **value** |
| Recomputes when? | Dependencies change |
| Use case | Expensive calculations, object/array memoization |
| Returns | **Value** (not a function or ref) |
| Common mistake | Using it unnecessarily (adds complexity with no gain) |

**⚠️ When *Not* to Use useMemo**

* Don’t overuse it. If your function is cheap or doesn’t cause re-renders, **you don’t need useMemo**.
* It's a performance optimization, not a functional necessity.

useCallback

useCallback is a React Hook that **returns a memoized version of a function**, preventing it from being re-created on every render—**unless its dependencies change**.

**🔹 Basic Syntax**

const memoizedCallback = useCallback(() => {

// function body

}, [dependencies]);

**🔸 Why Use useCallback?**

In React, **functions are re-created on every render** by default. This can cause:

1. **Unnecessary renders** in child components
2. Using useCallback helps **avoid those issues** by preserving the function identity.

**🧠 Example: Prevent Unnecessary Re-renders**

const Child = React.memo(({ onClick }) => {

console.log("Child rendered");

return <button onClick={onClick}>Click me</button>;

});

function Parent() {

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

const handleClick = useCallback(() => {

console.log("Button clicked");

}, []); // ✅ Same function reference across renders

return (

<div>

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

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

<Child onClick={handleClick} />

</div>

);

}

* Without useCallback, handleClick would be a new function each render.
* Child would re-render unnecessarily.

**🔁 useCallback vs useMemo**

| **Hook** | **Purpose** | **Returns** |
| --- | --- | --- |
| useMemo | Memoize a **value** | Value |
| useCallback | Memoize a **function** | Function |

**✅ React Hooks Comparison: useState, useEffect, useMemo, useRef, useCallback**

| **Hook** | **Purpose** | **Triggers Re-render?** | **Returns** | **Main Use Cases** |
| --- | --- | --- | --- | --- |
| **useState** | Store and manage **stateful data** | ✅ Yes | Current state + updater | Local component state (e.g. form inputs, counters) |
| **useEffect** | Run **side effects** after render | ❌ No (but can cause one) | Nothing (used for side-effects) | Data fetching. |
| **useMemo** | **Memoize a computed value** | ❌ No | **Value** | Avoid expensive recalculations |
| **useCallback** | Memoize a **function** | ❌ No | **Function** | Prevent function recreation, optimize with React.memo |
| **useRef** | Persist **mutable value or DOM ref** | ❌ No | .current object | Access DOM nodes, store values across renders (e.g. timers) |

**🔍 Quick Examples**

**useState**

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

**useEffect**

useEffect(() => {

document.title = `Count: ${count}`;

}, [count]);

**useMemo**

const expensiveValue = useMemo(() => computeHeavyTask(num), [num]);

**useRef**

const inputRef = useRef(null);

<input ref={inputRef} />

**useCallback**

const handleClick = useCallback(() => {

console.log("clicked");

}, []);

**🔄 When to Use What?**

| **Scenario** | **Hook to Use** |
| --- | --- |
| Track user input or UI changes | useState |
| Run code on mount, update, or unmount | useEffect |
| Prevent re-running a slow calculation | useMemo |
| Access DOM elements or store mutable values | useRef |
| Avoid function re-creation and unnecessary renders | useCallback |

**⚠️ Performance Tips**

* Don’t **overuse** useMemo or useCallback—they **add complexity** and don’t always improve performance.
* Use useRef for **non-UI** variables that you want to persist across renders without causing a re-render.

**React Router Dom**

**What is React Router DOM?**

**React Router DOM** is a **standard routing library** for **React.js** applications used to handle navigation between different components or "pages" in a single-page application (SPA).

**✅ Why do we need it?**

In a traditional website:

* Navigating to a new page means the browser reloads and requests a new HTML file from the server.

In a **React SPA**:

* We don’t want the whole page to reload.
* Instead, we want to **dynamically load different components** based on the URL — **without refreshing the page**.

**📦 Installing React Router**

npm install react-router-dom

**📘 Summary**

| **Feature** | **Purpose** |
| --- | --- |
| BrowserRouter | Enables routing |
| Route | Maps a path to a component |
| Routes | Wraps all Route components |
| Link | Navigate without reloading the page |
| useNavigate | Navigate programmatically |
| useParams | Access dynamic parts of the URL |

**Conditional UI**

**Conditional UI rendering** in React refers to displaying different user interface (UI) elements based on certain **conditions** — like user actions, data values, or app states.

Instead of showing everything all the time, you render **only what's needed** based on a condition.

import React, { useState } from "react";

const Pop = () => {

  const [open, setOpen] = useState(false);

  function pop() {

    setOpen(true);

  }

  return (

    <div>

      <div className="p-4">

        Lorem ipsum dolor sit amet consectetur adipisicing elit. Maxime corrupti

        doloribus culpa repellendus aliquid est mollitia dignissimos quod

        laborum, eaque quasi quam asperiores dolorem, nesciunt, laudantium

        veritatis. Nostrum, quisquam quos?

      </div>

      <button

        onClick={pop}

        className="bg-red-500 px-4 py-2 text-white rounded-2xl"

      >

        Press Me

      </button>

      {open && (

        <div className="fixed inset-0 flex justify-center items-center bg-black bg-opacity-50 z-50">

          <div className="bg-white p-6 rounded-lg shadow-lg">

            <p className="mb-4">Add card</p>

            <button

              onClick={() => setOpen(false)}

              className="bg-blue-500 text-white px-4 py-2 rounded"

            >

              Close

            </button>

          </div>

        </div>

      )}

    </div>

  );

};

export default Pop;

**Skeleton Loader**

A **skeleton loader** is a placeholder UI that mimics the structure of content that is being loaded — instead of showing a blank screen or a spinner, it shows "gray boxes" or shapes that look like the content layout.

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

const SkeletonCard = () => {

return (

<div className="w-64 h-32 bg-gray-300 animate-pulse rounded-md mb-4"></div>

);

};

const Card = ({ user }) => {

return (

<div className="w-64 h-32 bg-white shadow-md rounded-md p-4 mb-4">

<h3 className="text-lg font-bold">{user.name}</h3>

<p>{user.email}</p>

</div>

);

};

const UserList = () => {

const [loading, setLoading] = useState(true);

const [users, setUsers] = useState([]);

// Simulate API call

useEffect(() => {

setTimeout(() => {

setUsers([

{ name: 'Alice', email: 'alice@example.com' },

{ name: 'Bob', email: 'bob@example.com' },

]);

setLoading(false);

}, 3000); // 3 seconds loading

}, []);

return (

<div className="flex flex-col items-center mt-10">

{loading

? // Show 2 skeleton cards while loading

[1, 2].map((\_, i) => <SkeletonCard key={i} />)

: users.map((user, i) => <Card key={i} user={user} />)}

</div>

);

};

export default UserList;

**🔍 What's Happening?**

* SkeletonCard is a gray rectangle with animate-pulse, making it pulse like it's "loading".
* loading state determines whether to show skeletons or real data.
* On mount, a timeout simulates a delayed API call.

**🧠 Tips**

* For more realism, you can create skeleton shapes that match actual content layout: image circles, lines, etc.
* Use libraries like:
  + [**react-loading-skeleton**](https://github.com/dvtng/react-loading-skeleton?utm_source=chatgpt.com)
  + [**@mui/material Skeleton**](https://mui.com/material-ui/react-skeleton/)
  + [**React Content Loader (SVG-based)**](https://skeletonreact.com/)

**Error Handling**

Error handling in **React** is about managing and responding to errors that occur during rendering, in lifecycle methods, and in event handlers. Let's break this down clearly and with examples.

**🧠 Why is Error Handling Important in React?**

React components can fail due to:

* JavaScript errors in rendering
* API failures
* Uncaught exceptions in event handlers or lifecycle methods

Without proper error handling, your app can crash, show blank pages, or behave unexpectedly.

**Try/Catch in Event Handlers**

For **event handlers**, you must handle errors manually using try/catch.

// Error.js

import React, { useState } from 'react';

const Error = () => {

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

  const handleClick = () => {

    try {

      // Simulate an error

      throw new Error('Something went wrong!');

    } catch (err) {

      console.error('Caught error:', err);

      setError(err.message); // Store the error in state

    }

  };

  return (

    <div>

      <button onClick={handleClick}>Trigger Error</button>

      {error && <div style={{ color: 'red' }}>Error: {error}</div>}

    </div>

  );

};

export default Error;

**Loader State**

**✅ What is a Loader State?**

It's typically a boolean (true or false) stored in state that controls whether a **loading indicator (spinner, text, etc.)** is shown to the user.

import React, { useState } from 'react';

const LoaderExample = () => {

const [loading, setLoading] = useState(false);

const handleClick = () => {

setLoading(true); // Show loader

// Simulate a network request or delay

setTimeout(() => {

// Task is done

setLoading(false); // Hide loader

alert('Done!');

}, 2000);

};

return (

<div>

<button onClick={handleClick} disabled={loading}>

{loading ? 'Loading...' : 'Click Me'}

</button>

</div>

);

};

export default LoaderExample;

**👆 What’s happening here?**

* loading is the loader state.
* When the button is clicked:
  1. setLoading(true) is called to show a loading message or spinner.
  2. A setTimeout simulates a delay (e.g., API call).
  3. After 2 seconds, setLoading(false) hides the loader and enables the button again.

**🧠 When do you use a loader?**

* Fetching data from an API (e.g., fetch, axios)
* Submitting a form
* Waiting for an animation or transition

**Pagination**

**🧠 What is Pagination?**

**Pagination** is a way to **split a large set of data into smaller chunks (pages)** so it's easier to view and manage.

Think of it like this:

You have **1000 books**, but you only want to show **10 books per page**.  
So instead of showing all 1000 books at once, you show:

* Page 1: Books 1–10
* Page 2: Books 11–20
* Page 3: Books 21–30

**✅ Why use Pagination?**

* ✅ Improves performance (less data to load at once)
* ✅ Easier for users to read
* ✅ Saves space on the page

**📚 Real-life Examples**

You see pagination in many apps:

* **Google Search** → shows page numbers at the bottom
* **E-commerce websites** → next/prev buttons to browse products
* **Blog sites** → list of articles with page navigation

**💻 Example in React (Basic Client-side Pagination)**

Let’s say you have 50 items and want to show 5 per page.

import React, { useState } from 'react';

const PaginationExample = () => {

const items = Array.from({ length: 50 }, (\_, i) => `Item ${i + 1}`);

const [currentPage, setCurrentPage] = useState(1);

const itemsPerPage = 5;

// Calculate which items to show

const startIndex = (currentPage - 1) \* itemsPerPage;

const endIndex = startIndex + itemsPerPage;

const currentItems = items.slice(startIndex, endIndex);

const totalPages = Math.ceil(items.length / itemsPerPage);

return (

<div>

<h2>Pagination Example</h2>

<ul>

{currentItems.map((item) => (

<li key={item}>{item}</li>

))}

</ul>

<div>

<button

onClick={() => setCurrentPage((p) => Math.max(p - 1, 1))}

disabled={currentPage === 1}

>

Prev

</button>

<span> Page {currentPage} of {totalPages} </span>

<button

onClick={() => setCurrentPage((p) => Math.min(p + 1, totalPages))}

disabled={currentPage === totalPages}

>

Next

</button>

</div>

</div>

);

};

export default PaginationExample;

**🔍 What's Happening?**

* items is your full list (50 items)
* currentPage tracks the current page
* itemsPerPage is how many to show
* You calculate which items to show using .slice()
* "Prev" and "Next" buttons allow moving between pages

**🔁 Server-side vs Client-side Pagination**

| **Type** | **Where it happens** | **Best for** |
| --- | --- | --- |
| Client-side | In the browser | Small datasets |
| Server-side | On the backend/API | Large datasets (e.g., 10,000+ records) |

**Search Bar**

**🔍 What is a Search Bar?**

A **search bar** is a box where users can **type text** to **find specific items** from a larger list.

Just like Google: you type what you're looking for, and it shows matching results.

**💻 How It Works (Simple React Example)**

Here's a very basic example of how to build a search bar in React:

import React, { useState } from 'react';

const SearchBarExample = () => {

const [searchTerm, setSearchTerm] = useState('');

const fruits = [

'Apple', 'Banana', 'Orange', 'Mango',

'Grapes', 'Kiwi', 'Pineapple', 'Lemon',

'Guava', 'Papaya'

];

// Filter the list based on searchTerm

const filteredFruits = fruits.filter(fruit =>

fruit.toLowerCase().includes(searchTerm.toLowerCase())

);

return (

<div>

<h2>Fruit Search</h2>

<input

type="text"

placeholder="Search fruits..."

value={searchTerm}

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

/>

<ul>

{filteredFruits.map(fruit => (

<li key={fruit}>{fruit}</li>

))}

</ul>

</div>

);

};

export default SearchBarExample;

**🔁 User Flow**

1. Page shows all fruits.
2. User types something like "ap" into the search bar.
3. filteredFruits becomes a smaller list.
4. Only matching fruits are shown.

**🧠 Summary**

* A **search bar** helps users find things quickly.
* You store the **typed value in state**.
* You **filter the list** based on that value.
* You **display the filtered results**.

**React CRUD Operation**

CRUD stands for **Create, Read, Update, Delete** — the four basic operations for managing data in an application. In the context of **React.js**, CRUD operations are typically implemented in combination with a backend API (often RESTful or GraphQL) and React state management (like useState, useEffect, or context).

**1. Create (POST)**

**Goal:** Add new data (e.g., a new user, post, product).

**Example:**

import axios from 'axios';

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

const handleCreate = async () => {

await axios.post("https://api.example.com/users", { name });

};

**2. Read (GET)**

**Goal:** Fetch and display existing data.

**Example:**

import axios from 'axios';

const [users, setUsers] = useState([]);

useEffect(() => {

const fetchUsers = async () => {

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

setUsers(response.data);

};

fetchUsers();

}, []);

**3. Update (PUT or PATCH)**

**Goal:** Modify existing data (e.g., editing a user profile).

**Example:**

import axios from 'axios';

const handleUpdate = async (id, updatedName) => {

await axios.put(`https://api.example.com/users/${id}`, { name: updatedName });

};

**4. Delete (DELETE)**

**Goal:** Remove data (e.g., deleting a user or a post).

**Example:**

import axios from 'axios';

const handleDelete = async (id) => {

await axios.delete(`https://api.example.com/users/${id}`);

};**Summary of CRUD in React:**

| **Operation** | **HTTP Method** | **React Involvement** |
| --- | --- | --- |
| Create | POST | Form input + fetch |
| Read | GET | useEffect + fetch |
| Update | PUT/PATCH | Form + fetch with ID |
| Delete | DELETE | Button + fetch with ID |