React js NOTES

React js :

- a popular open-source JavaScript library.
- used for building user interfaces, particularly for single-page applications where you need to update the view dynamically without reloading the page .
- developed and maintained by Facebook .
- High demand due to faster speed .
- Large community support .
- faster and easier to learn .
- allows developers
    - to create reusable UI **components**
    - manage the state of an application and
    - handle dynamic data efficiently .

NOTE - A **Single-Page Application (SPA)** is a type of web application that dynamically rewrites the current page rather than loading entire new pages from the server. This means that the web page remains static while only specific parts of the page are updated as the user interacts with the application.

A **Single-Page Application (SPA)** and a **One-Page Website** might sound similar, but they serve different purposes and are implemented in distinct ways.

A one-page website is a **static webpage** that contains all its content on a single HTML page. Users can scroll through the content or navigate using anchor links, but the entire site is contained within a single page, often with minimal interactivity.

One-page websites are often used for **simple, informational purposes** such as portfolios, landing pages, event pages, or small business websites where the focus is on displaying a limited amount of content or driving a specific call-to-action.

React creates user interface using components

- In React.js, **components** are the building blocks of the user interface.
- A React application is composed of multiple components , each responsible for rendering a specific part of the UI.
- Components make it easier to manage, reuse, and organize code in complex applications.
- Component can be as small as a button or as large as an entire page

React uses virtual DOM

- The **Virtual DOM (Document Object Model)** is a core concept in React.js that significantly enhances the performance and efficiency of web applications .
- It is one of the key reasons React is fast and responsive, as it optimizes the process of updating the UI when the state of the application changes.

What is the Virtual DOM ?

- The **DOM** is a programming interface for web documents, representing the structure of the webpage as a tree of objects/nodes , where each node is an element on the page (such as `<div>`, `<h1>`, etc.) .
- Whenever the UI changes, the DOM needs to be updated to reflect those changes. However, updating the actual DOM (often called the "real DOM") can be slow because it requires recalculating styles, layouts, and repainting parts of the page .
- React addresses this inefficiency by using a **Virtual DOM**. The Virtual DOM is an in-memory, lightweight representation of the actual DOM. React keeps this virtual copy in sync with the real DOM and only updates the real DOM when necessary.

How the Virtual DOM Works:

1. **Initial Rendering**: (Render → Display)
    - When a React component renders for the first time , React creates a **virtual DOM tree** based on the JSX code.
2. **State or Props Change**:
    - When a change occurs in the component's state or props, React re-renders the component and creates a new virtual DOM tree that reflects these changes.
3. **Diffing Algorithm**:
    - React compares the new virtual DOM tree with the previous one using an efficient **diffing algorithm**. This process is called **reconciliation**.
    - React determines what has changed by performing a "tree diff" between the old and new virtual DOM, detecting changes at the component level and down to individual elements.
4. **Minimal DOM Updates**:
    - Once the changes are identified, React updates only the parts of the actual DOM that have changed, rather than re-rendering the entire DOM. This minimizes the number of manipulations needed and significantly improves performance.

NOTE - React , angular and view js all are used to build single page applications . But React is most common among them and with a large community .

—> React and jQuery are libraries .

—> angular , vue and next js are frameworks .

Companies using React JS :

- Facebook
- Netflix
- Flipkart
- Myntra
- Pinterest etc.

React js can work on different types of files which includes .js , .jsx , .Ts and .Tsx .

- Jsx —> js + xml (It allows developers to write HTML-like code directly within JavaScript .) JSX stands for javascript syntax extension .

Components in react JS

- In React.js, **components** are the building blocks of the user interface.
- A React application is composed of multiple components, each responsible for rendering a specific part of the UI .
- Components make it easier to manage, reuse, and organize code in complex applications .
- The same component can be used at different locations , so we can create components and can call it many times .
- Component is a reusable and independent piece of code .

Types of Components in React:

React components are mainly divided into two types:

1. **Functional Components** (also called Stateless Components (Now they are stateful)
2. **Class Components** (also called Stateful Components)

1. **Functional Components**:

- **Definition**: Functional components are simple JavaScript functions that accept props as an argument and return JSX (the UI) .
- **Stateless or Stateful**: Initially, functional components were stateless, but with the introduction of React **Hooks**, they can now handle state and side effects.
- **Preferred Usage**: They are simpler, easier to read, and more concise compared to class components.

Syntax :

```
function Welcome ()
 {
  return ( <h1>Hello world</h1> );      //Parenthesis after return is optional
}
```

NOTE - If we are using more than 1 tag in the component , then they all must be in the single parent tag . (or can be in an empty tag <> —------ </>)

E.g

```
function Welcome()
 {
  return ( <div className=”parent”>   //Use className instead of class
<h1>Hello world .</h1>
<h5>I am vedant .</h5>
<p>Hope you are all doing well . Here I am using more than 1 tag inside in a single div with className=”parent” .</p>
</div> ) ;
}
```

NOTE - In React js we give class names using className attribute .

NOTE - One can also create a functional component in a separate .jsx file and can import that component in the current file .

E.g

In Header.jsx (**File name need not to be the component name and its 1st letter should be in capital case**)

function Header() (**Name of the component should be start with capital letter**)

{ return (<h1>Hello world</h1>) ; }

export default Header ;

(Here we are using default export ,component must be exported from the file)

In App.js

import Header from ' . / Header '; (component must be imported)

function App() {

return (

<Header/>

 ); }

NOTE : One can create a component file and can use the shortcut key .

'rafce' to import the boilerplate that creates the functional component with the same name as the name of the file .

One can also render Header component like - <Header></Header>

NOTE : All the tags used in a jsx file must be closed .

Example : we can't use <img> tag to display image . It must be <img/>

or <img></img>

Whenever we create a component by creating a file , we must import React like - import { React } from ‘react ;

‘rafce’ will automatically import react .

Class component in react

A **class component** in React is one of the two main ways to define a component . It's a JavaScript ES6 class that extends `React.Component` and must include a `render()` method that returns JSX.

✅ Basic Example of a Class Component:

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

class Welcome extends Component {
  render() {
    return <h1>Hello World !</h1>;
  }
}
export default Welcome;
```

Exports and its types

In React (and JavaScript in general), **exports** allows you to share code between different files or modules. There are two types of exports:

1. **Default Exports**
2. **Named Exports**

Both of these are essential when organizing your React code into reusable components and utility functions. Here's an explanation of how each type works:

1. Default Exports

A **default export** allows you to export a single value, function, or component from a file. When importing a default export, you can choose any name for the import.

Syntax for Default Export:

```
// Exporting a default value (a component in this case)

function Button () {

return ( return <button>Click Me</button> ) ; }

export default Button ;
```

OR

```
export default function Button() {

  return <button>Click Me</button>;  }
```

Importing a Default Export:

```
// You can import it using any name

import MyButton from './Button';

function App() {

  return <MyButton />;

}
```

Here:

- `Button` is exported as the default export from `Button.js`.
- When importing, we can name it anything (`MyButton` in this case).

Default Export Rules:

- Only **one** default export is allowed per file.
- You don’t need to use curly braces when importing a default export.

2. Named Exports

A **named export** allows you to export multiple values or components from a single file. When importing a named export , you must use the exact name of the export inside curly braces .

Syntax for Named Export:

```
// Exporting multiple components or values
In Layout.jsx
export function Header() {
  return <h1>This is the header</h1>;
}

export function Footer() {
  return <h1>This is the footer</h1>;
}
```

OR

```
Header() {
    return <h1>This is the header</h1>;
}
Footer() {
    return <h1>This is the footer</h1>;
}
export {Header,Footer}
```

Importing Named Exports:

```
// Importing named exports with curly braces
In App.js
import { Header, Footer } from './Layout';
```

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

Here:

- `Header` and `Footer` are named exports from `Layout.js`.
- When importing, we use curly braces `{}` and must use the exact names `Header` and `Footer`.

Named Export Rules:

- You can have **multiple named exports** per file.
- When importing, you must use the same name as the exported value.
- You can also **alias** named exports while importing (i.e., change the name) .

What is Babel ?

Babel is a **JavaScript compiler** that helps convert modern JavaScript (ES6+) code into older versions (ES5 or earlier) so that it can run in older browsers or environments that do not support the latest JavaScript features.

Why Use Babel ?

✅ **Ensures Compatibility** – Older browsers might not support new JavaScript features like `let`, `const`, arrow functions, or optional chaining. Babel translates them into compatible code.
✅ **Supports Modern JavaScript** – You can write ES6, ES7, and beyond without worrying about browser support.

✅ **Works with Frameworks** – Many libraries and frameworks (like React) use Babel to enable modern JavaScript syntax.

Babel with React

Babel is often used in React to convert JSX into regular JavaScript.

Example (JSX) :

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

After Babel Transpilation :

const element = React.createElement("h1", null, "Hello, world!");

NOTE : Babel dependencies install along with other dependencies we run - npm install command

React fragment :

A **React Fragment** is a special component that lets you group multiple elements without adding extra nodes to the DOM. This is useful when you want to return multiple elements from a component without introducing an additional wrapper element (like a `<div>`), which can sometimes mess with styling or layout.

Key Points

- **No Extra Markup:** Fragments do not render any additional elements in the DOM.
- **Cleaner HTML:** Helps in keeping the HTML structure clean and semantically correct.
- **Syntax:** You can use `<React.Fragment>` or the shorthand `<>...</>`.

Example Using `<React.Fragment>` :

```
function MyComponent() {
  return (
    <React.Fragment>
      <h1>Title</h1>
      <p>Description text.</p>
    </React.Fragment>
  );
}
```

Example Using Shorthand Syntax :

```
function MyComponent()
 {
  return (
    <>
      <h1>Title</h1>
      <p>Description text.</p>
    </>
  );
}
```

When to Use Fragments

- Multiple Sibling Elements : When a component needs to return multiple sibling elements.
- Avoiding Unnecessary Wrappers : When adding extra wrappers might affect styling or layout.
- Improved Performance : By not adding unnecessary nodes, it keeps the DOM cleaner, which can help with performance in larger applications.

In summary , React Fragments help you write cleaner and more maintainable React components by grouping elements without adding extra nodes to the rendered output .

JavaScript in JSX with Curly Braces :

JSX lets you write HTML-like markup inside a JavaScript file, keeping rendering logic and content in the same place. Sometimes you will want to add a little JavaScript logic or reference a dynamic property inside that markup. In this situation, you can use curly braces in your JSX to open a window to JavaScript.

Example :

```
export default function Avatar() {
  const avatar = 'https://i.imgur.com/7vQD0fPs.jpg';
  const description = 'Gregorio Y. Zara';
  return (
    <img
      className="avatar"
      src={avatar}
      alt={description}  />
  );
}
```

Install Node js

Installing Node.js is important for running a React app because React relies on tools and libraries that require Node.js and npm (Node Package Manager). Here's why Node.js is essential:

Create React App (CRA) : The most common way to set up a React project is by using the `create-react-app` command , which depends on Node.js to execute its commands.

Development Server : React uses tools like Webpack or Vite, which require Node.js to serve the app during development. Node provides the runtime for these tools to work.

npm (Node Package Manager) : Node.js comes bundled with npm, which is used to install , manage, and update the dependencies required by your React application .

- For example, libraries like `react`, `react-dom` , `axios`, or `redux` are installed via npm.

Node allows you to run js other than browser .

Usually the V8 engine from the browser runs the js code . When Node.js was created, it needed an engine to execute JavaScript outside the browser. V8 was chosen because of its **Performance** , **Open Source ,Embeddability etc .**

Node.js is a free, open-source, cross-platform JavaScript runtime environment that lets developers create servers, web apps, command line tools and scripts.

As soon as we download the node in our system . It installs the npm and npx package manager in our system .

One can check the versions of installed node and npm , npx by running following commands in the terminal

node -v

npm -v

npx -v

NOTE : As we know node helps to run js code outside the browser , so to run the js file outside the browser , install the node and run the following command .

node file_name // This will run the js file in the terminal

npm (Node Package Manager) :

npm is a library of packages .

npm is the default package manager for Node.js. It is used to:

- Install, manage, and update JavaScript libraries and dependencies .
- Manage the `node_modules` folder, where installed packages are stored.
- Publish and share JavaScript packages with the developer community.

Key Features of npm :

1. **Install Packages** :

- Install local or global packages from the npm registry.

npm install <package-name> # Installs a package locally

npm install -g <package-name> # Installs a package globally

2. **Manage Dependencies**:

- Saves dependencies in the `package.json` file.

npm install <package-name> --save

npm install <package-name> --save-dev

3. **Run Scripts**:

- Define custom scripts in `package.json` and run them using npm.

npm run <script-name>

npx (Node Package Executor) :

npx is a tool bundled with npm (since npm version 5.2.0) that allows you to:

- Execute Node.js packages and binaries directly without installing them globally .
- Simplify running one-off commands or temporary tools.

Key Features of npx :

1. **Execute Packages Without Installation**:
    - Run a package directly from the npm registry.

npx <package-name>

Example :

npx create-react-app my-app

npm and npx both are package managers . npm is used when we want to install and manage packages by installing it in our system and npx is used when we want to execute packages directly without installing .

Practical Examples

Using npm to Install and Run

If you want to create a React app:

1. Install the package globally:
   `npm install -g create-react-app`
2. Use the tool:
   `create-react-app my-app`

Using npx to Skip Installation

You can skip global installation with npx:

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

- npx will download the tools temporarily , use it , and clean up afterward.

Now-a-days we don't use the create-react-app `command, instead we` use tools like vite to run the react project .

What is Vite ?

Vite (French for "fast") is a **modern frontend build tool** that provides a faster and more optimized development experience compared to traditional tools like Create React App (CRA) .

How to Create a React Project with Vite?

1️⃣ Open the terminal and run:

```
npm create vite@latest my-app(project-name)
```

2️⃣ Navigate to the project:

```
cd my-app
```

3️⃣ Install dependencies:

```
npm install
```

4️⃣ Start the development server:

```
npm run dev
```

5️⃣ Open the browser and go to **http://localhost:5173/** (default Vite port).

file structure created by `create-react-app:`

```
my-app
├── build
├── node_modules
├── public
│   ├── favicon.ico
│   ├── index.html
│   └── manifest.json
├── src
│   ├── App.css
│   ├── App.js
│   ├── App.test.js
│   ├── index.css
│   ├── index.js
│   ├── logo.svg
│   └── serviceWorker.js
├── .gitignore
├── package.json
└── README.md
```

Folder and File Descriptions

1.Root Directory

- **README.md**: Contains project details and instructions for setup.
- **package.json**: Lists project dependencies, scripts, and metadata.
- **.gitignore**: Specifies files and folders ignored by Git (e.g., node_modules).
- **yarn.lock / package-lock.json**: Ensures consistent dependency versions.

2. public /

Files in this folder are served as-is without Webpack processing.

- **`index.html`**: The entry point of the app where React mounts.
- **`favicon.ico`**: The browser tab icon.
- **Static Assets**: Any static files like images or meta tags.

3. src /

The main directory where application code resides.

i. `App.js`

The root React component where the application is structured. Contains the layout and routing logic .

ii. `index.js`

The entry point of the React app. It renders the `<App />` component into the DOM .

iii. `index.css`

Global CSS styles applied across the application.

The **`src/`** folder in a React app is the primary directory where all the application code resides. This folder contains components, styles, hooks, utilities, and other resources necessary for building and structuring the app. Below is a detailed overview of the typical structure and contents of the **`src/`** folder in a React application.

Structure of the `src/` Folder :

src/

├── assets/

├── components/

├── context/

├── hooks/

├── pages/

├── services/

├── utils/

├── App.css

├── App.js

├── index.css

├── index.js

4 . node_modules /

The **`node_modules`** folder in a React app (or any Node.js-based project) is a directory where all the project dependencies are stored after running `npm install` or `yarn install` .

Key Points About the `node_modules` Folder :

Purpose:

- It contains the JavaScript libraries and packages that your project depends on, including any sub-dependencies of those libraries.

Automatic Management:

- You don't create or modify the `node_modules` folder manually. It is generated automatically based on the dependencies listed in the **`package.json`** file.

Dependencies Included:

- Direct dependencies: Packages you explicitly include in your project.
- Transitive dependencies: Sub-dependencies required by the packages you installed.

Size:

- The `node_modules` folder can grow very large, as it includes every package your project needs, including redundant or deeply nested dependencies.

Not Committed to Version Control:

- The `node_modules` folder is excluded from version control systems (e.g., Git) via the `.gitignore` file. This is because it can be regenerated using `npm install` or `yarn install` based on the `package.json` and `package-lock.json`/`yarn.lock` files.

NOTE : One can implement react project by creating any of the following files

.js file / .jsx file / .Ts file / .Tsx file

Jsx file —> javascript + xml (combination of javascript and html code)

JSX stands for javascript syntax extension .

SetUp and adding bootstrap to react project

1.install the bootstrap packages into the react app by running the following command in the terminal of the project folder in which you are creating a react app

npm i bootstrap

2.import some inbuilt css and js bootstrap files in the index.js file

import “bootstrap/dist/css/bootstrap.css”
import “bootstrap/dist/js/bootstrap.bundle”

Now One can use all the bootstrap classes and components in the react project .

E.g container class , text-danger , text-primary etc

Reactive Bootstrap :

Reactive Bootstrap refers to the integration of Bootstrap, a popular front-end framework, with reactive frameworks like **React**, **Vue**, or **Angular**. This combination allows developers to build responsive, interactive, and reactive web applications using Bootstrap's pre-designed components and styles while leveraging the dynamic capabilities of these frameworks.

Bootstrap with React :

- Bootstrap does not natively support React, but libraries like **React-Bootstrap** or **reactstrap** adapt Bootstrap components for React applications.
- These libraries provide ready-to-use React components that adhere to Bootstrap's design and functionality.

React-Bootstrap :

- A library that offers a complete re-implementation of Bootstrap components as React components.

Installation: npm install react-bootstrap bootstrap

React bootstrap provides built-in bootstrap components such as Container component .As we know container is a class in bootstrap but react-bootstrap Container is a component with ‘C’ capital .

Example :

```
import React from ' react ' ;

import { Container } from 'react-bootstrap';

function App() {

  return (

    <Container>

      <h1>Hello, World!</h1>

      <p>This content is inside a React-Bootstrap Container.</p>

    </Container>

  );}
```

NOTE : Use react.bootstrap website to copy react-bootstrap component

Example :

```
import React from 'react' ;

import 'bootstrap/dist/css/bootstrap.min.css' ;

import Button from 'react-bootstrap/Button' ;
```

```
function App () {

 return <Button variant="primary">Click Me!</Button>;

 }

export default App ;
```

Adding Images in Component :

To add an image in a component we can use <img/> tag , but we can't provide the source or the path of an image .
First we have to insert that image file in a folder .
And import that image file in the current component file like :

```
import img1 from ../Image/imageName
```

Here i store the image by creating a folder named 'Image'
And importing that image in the variable by providing the image file path .
Here the image is stored in img1
Now we can use that variable name in the src attribute of <img> tag .
Example :

```
<img src={img1} >
```

As we are using variable names we must use curly brackets .

Props in React :

Props in React (short for "properties") are a mechanism for passing data from one component to another, typically from a parent component to a child component. They allow components to be dynamic and reusable by customizing their behavior or display based on the data provided .

Key Features of Props in React

1. **Read-Only**:
    - Props are immutable, meaning they cannot be modified by the child component that receives them .
    - If dynamic behavior is needed, the parent component must manage the state and pass updated values as props .
2. **Unidirectional Flow**:
    - Data flows from parent to child, ensuring a predictable structure in your application.
3. **Customizable Components**:
    - Props allow the same component to be reused with different data or configurations. (Very IMP point)
4. **Type Checking**:
    - Using libraries like **PropTypes** or **TypeScript**, you can define and validate the types of props passed to components .

How Props Work :

1. Passing Props

Props are passed to a child component in the same way you pass attributes to HTML elements.

```
function ParentComponent()

 {

  return <ChildComponent name="John" age={25} />;  → use {} to set number

}

function ChildComponent(props) {

  return (

    <div>
```

```
      <p>Name: {props.name}</p>
      <p>Age: {props.age}</p>
    </div>
  ); }
```

OR

```
function ChildComponent({name,age}) {
  return (
    <div>
      <p>Name: {name}</p>
      <p>Age: {age}</p>
    </div>
  );
```

NOTE : One can also pass default values as props also

```
function ChildComponent({name=”Veda” ,age={25} }) {
  return (
    <div>
      <p>Name: {name}</p>
      <p>Age: {age}</p>
    </div>
```

NOTE : Props store the passed data as javascript Objects in the form of key-value pairs .

NOTE : All the content inside the curly braces { } in jsx file will be treated as javascript logic .

Example : if we use {2+3} in a jsx file along with html , it will result in 5 .

Example : If we pass Name=”Vedant” and Age={24} in the parent Component

then props will be

Props =

{

‘Name’ : ‘vedant’,

‘Age’ : 24

}

Here props is the name of the object which is a user defined name .

One can access the props data in the child component same as we access values of objects i.e by using keys like { Props.Name } , { Props.Age } .

One can also pass arrays and objects as props .

Props.children :

In React, `props.children` is a special property of `props` that allows you to pass child elements directly into a component, making it possible to compose components with nested structures.

How It Works

When you wrap elements or components inside a custom component's tags, those elements become available in the `props.children` property of the custom component.

Example

Basic Usage

```
function App()
{
  return (
    <Wrapper>
      <h1>Hello, World!</h1>
      <p>This is wrapped content.</p>
    </Wrapper>
  );
}

function Wrapper(props)
{
  return <div> {props.children} </div>;
}
```

What Happens Here ?

- The `h1` and `p` tags are passed as `props.children` to the `Wrapper` component.
- Inside the `Wrapper` component, `props.children` renders these elements inside the `div`.

Functions as props :

In React, you can pass **functions as props** to a child component. This pattern is often used for communication between components or to provide specific behaviors (like event handlers) from a parent component to a child.

Why Pass Functions as Props ?

1. **Parent-Child Communication**:
    - Allows child components to call a function defined in the parent, enabling data or event handling in the parent .
2. **Custom Behaviors**:
    - Pass a function to customize the behavior of a reusable child component.
3. **Improves Reusability**:
    - Parent components can define the logic, and child components execute it via the function.

Basic Example: Handling Button Clicks

Parent Component :

```
import React from "react";

import Button from "./Button";

function App() {

  const handleClick = () => {

    alert("Button clicked!");   };

  return (

    <div>

      <h1>Props as Functions Example</h1>

      <Button fun1={handleClick} />

    </div>

  );  }

 export default App;
```

Child Component (`Button`) :

```
function Button({ fun1 })

 {

  return <button onClick={fun1}> Click Me </button>;

}

export default Button;
```

How It Works:

- The `App` component defines the `handleClick` function.
- It passes `handleClick` to the `Button` component as the `fun1` prop.
- When the button is clicked, the `fun1` function is executed, triggering the `handleClick` function in the parent.

Hooks in React :

Hooks are special functions introduced in React **16.8** that let you use **state** and other React features in functional components. Before hooks, managing state and lifecycle in React required class components. With hooks, functional components can manage state, side effects, and other React features , making them more powerful and concise .

Why Use Hooks?

1. **Simpler Syntax** :
    - Hooks eliminate the need for class components in most cases.
2. **Reusable Logic** :
    - Extract stateful logic into **custom hooks** that can be reused across components .

3. **Avoid Complexity** :
    - Reduce the complexity of managing lifecycle methods by combining them in a single hook like `useEffect`.

Rules of Hooks

1. **Call Hooks at the Top Level**:
    - Do not call hooks inside loops, conditions, or nested functions.
2. **Call Hooks in React Functions Only**:
    - Hooks should only be called in functional components or custom hooks.

1. `useState` :

The `useState` hook is one of the most commonly used React hooks. It allows you to add **state** to a functional component. With `useState`, you can declare state variables and update them without needing a class component.

How `useState` Works

- **Syntax**:

const [state, setState] = useState(initialState);

- `state`: Name of the state that stores the current state value.
- `setState`: A function to update the state.
- `initialState`: The initial value of the state.

Example: Counter Component

import React, { useState } from "react";

function Counter() {

// Declare a state variable 'count' with an initial value of 0

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

return (

  <div>

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

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

    <button onClick={() => setCount(count - 1)}>Decrement</button>

  </div>

); }

export default Counter;
```

Explanation:

1. `useState(0)` initializes the `count` state variable with `0`.
2. `setCount` is used to update the `count` state when the button is clicked.
3. React re-renders the whole component when the state is updated . But the useState() statement renders only once , that's why the value of count keeps changing .

Updating State

1. **Replace State Directly**:

- Provide a new value to `setState`.

```
——--> setCount(10); // Sets count to 10
```

2. **Update State Based on Previous State**:

- Use a callback function when the new state depends on the previous state.

```
——-->setCount((prevCount) => prevCount + 1);
```

Re-rendering In React :

Re-rendering in React refers to the process where a component updates and renders again in the Virtual DOM.

Re-rendering in React means that a component updates and runs its render function again to update the UI. This happens when the component detects changes in **state**, **props**, or **context**.

This happens when:

1. **State Changes** – If a component's state is updated using `useState` or `useReducer`, React triggers a re-render.
2. **Props Change** – If a parent component passes new props to a child component, the child will re-render.
3. **Context Updates** – When a context value changes, all consuming components will re-render .

How to Optimize Re-Rendering

- **Use `React.memo()`**: Prevents unnecessary re-renders if props remain the same.
- **Use `useCallback()` and `useMemo()`**: Helps prevent child components from re-rendering unnecessarily.
- **Avoid Unnecessary State Updates**: Ensure that state updates only when required.

useEffect Hook :

The **`useEffect`** hook is one of the most commonly used hooks in React. It lets you perform **side effects** in functional components, such as:

- Fetching data from APIs.
- Subscribing to or cleaning up resources like timers or event listeners.
- Updating the DOM directly .

Syntax of `useEffect` :

```
useEffect(() => {
  // Side effect logic
  return () => {
    // Cleanup logic (optional)
  };
}, [dependencies]);
```

Dependencies in React

In React, **dependencies** refer to values (such as variables, state, props, or functions) that a component relies on to perform actions like rendering, effects, or memoization. They are commonly used in hooks like useEffect, useCallback, and useMemo.

Parameters of `useEffect`

1. **Callback Function**:
   - Contains the logic for the side effect (e.g., API call, DOM updates).
   - Can return a cleanup function if needed.
2. **Dependencies Array**:
   - A list of values that the effect depends on.
   - Determines when the effect should re-run.
   - If empty ([]), the effect runs only once (similar to componentDidMount).

Basic Usage of `useEffect`

1. Run Once on Mount

Perform an action when the component is first rendered.

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

function App() {
  useEffect(() => {
    console.log("Component mounted");
    return () => {
      console.log("Cleanup before component unmounts");
    };
  }, []); // Empty dependency array means this runs only once
  return <div>Hello, World!</div>;
}

export default App;
```

Using Dependencies in `useEffect`

2. Re-run on Dependency Change

The effect runs whenever a dependency changes.

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

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

```
    console.log(`Count updated to: ${count}`);
  }, [count]); // Runs whenever `count` changes
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={( ) => setCount(count + 1)}>Increment</button>
    </div>  );  }
export default Counter;
```

Cleanup in `useEffect`

3. Example: Clearing a Timer

Use cleanup logic to prevent memory leaks, such as clearing intervals or unsubscribing from events.

```
import React, { useState, useEffect } from "react";
function Timer() {
  const [seconds, setSeconds] = useState(0);
  useEffect(() => {
    const interval = setInterval(() => {
      setSeconds((prev) => prev + 1);
    }, 1000);
    return () => {
      clearInterval(interval); // Cleanup when the component unmounts
```

```
    };
  }, []); // Run only once
  return <div>Seconds: {seconds}</div>;
}
export default Timer;
```

NOTE : cleanUp function will run as soon as we unmount (remove) the value .

Using `useEffect` Without Dependencies

5. Run Effect After Every Render

If you omit the dependency array, the effect runs after every render (similar to `componentDidUpdate`) .

```
useEffect(() => {
  console.log("Effect runs after every render");
} );
```

⚠️ **Avoid this unless absolutely necessary**, as it may lead to performance issues.

Common Patterns

6. Dependent on Multiple Values

Run the effect when any value in the dependency array changes.

```
useEffect(() => {
  console.log("Effect triggered");
}, [value1, value2]);
```

Key Points About `useEffect`

1. **Dependency Array**:
    - Avoid unnecessary re-renders by specifying dependencies correctly.
    - Missing dependencies can cause stale values in your effect.
2. **Cleanup Function**:
    - Always clean up resources (e.g., intervals, subscriptions) in the return statement of the `useEffect` callback.
3. **Asynchronous Functions**:
    - `useEffect` cannot accept an async function directly, but you can define an async function inside it:

```
useEffect(() => {
  const fetchData = async () => {
    const result = await fetch("/api");
    // handle result
  };
  fetchData();
}, []);
```

When to Use `useEffect`

- Fetching or syncing data with an external source.
- Setting up or cleaning up event listeners or subscriptions.
- Managing timers or intervals.
- Performing animations or DOM updates outside React.

`useContext` Hook in React :

The `useContext` hook in React provides an easy way to access values from a **context** in functional components. It simplifies the process of consuming shared data, such as theme, authentication, or global settings, without having to pass props manually down the component tree.

How `useContext` Works

- **Context**: A way to share values (data) across components without having to pass props manually at every level.
- **`useContext`**: A React hook that allows components to access the current value of a context directly.

Steps to Use `useContext`

1. **Create a Context**: Use `React.createContext()` to create a context.
2. **Provide a Context Value**: Use the `Context.Provider` to wrap the component tree and supply the shared value.
3. **Consume the Context**: Use the `useContext` hook to access the value inside any child component.

Basic Example: Theme Context

1. Create and Provide a Context

```
import React, { createContext , useContext } from "react";

// Create a context
const ThemeContext = createContext();

function App ()
 {
```

```
  return (
    <ThemeContext.Provider value="dark">
      <Toolbar />
    </ThemeContext.Provider>
  );
}

function Toolbar() {
  return <ThemeButton />;
}

function ThemeButton() {
  // Use the context
  const theme = useContext(ThemeContext);
  return <button>{`Theme: ${theme}`}</button>;
}
export default App;
```

`ThemeContext.Provider` provides the value `"dark"`.
`useContext(ThemeContext)` allows `ThemeButton` to access the value `"dark"` without passing it down as a prop.

Key Points About `useContext`

1. **Simplifies Context Usage** :
    - Avoids the need for nested render props or `Context.Consumer`.
2. **Use with `Context.Provider`**:
    - `useContext` must be used within a matching `Provider`. Otherwise, it returns the default value set in `createContext`.
3. **Triggering Re-renders** :
    - A component consuming a context re-renders whenever the context value changes.

4. **Avoid Overusing Context** :
    - Use context for truly shared state (e.g., themes, authentication).
    - For large-scale state management, consider libraries like Redux or Zustand.

When to Use `useContext`

- To share global or app-wide data like:
    - Theme (dark/light mode).
    - Authentication state (logged-in user).
    - Language preferences.
    - Global settings or configurations.

State lifting :

State lifting in React refers to the process of moving the state up the component hierarchy to a common ancestor so that multiple child components can share and synchronize the same state.

This approach is essential when two or more components need to share data or communicate with each other.

Why Lift State?

1. **Shared State**: When multiple components need access to the same data, lifting state avoids redundancy and keeps the state centralized.
2. **Unidirectional Data Flow**: React uses a one-way data flow, meaning parent components pass data to children. Lifting state ensures that data changes propagate down the component tree correctly.
3. **Simplifies Communication**: Child components can share information indirectly by modifying the shared state in their parent.

How to Lift State

Scenario: Synchronizing Two Inputs

Imagine a scenario where two child components need to share and synchronize their input values .

Step-by-Step Implementation

1. **Create a Parent Component**:
    - Define the shared state in the parent component.
    - Pass the state and updater functions as props to the children.
2. **Child Components**:
    - Use the props passed from the parent to display or update the state.

Code Example

Parent Component:

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

import Input from "./Input";

function App() {

  const [sharedValue, setSharedValue] = useState("");

  return (

    <div>

      <h1>State Lifting Example</h1>

      <Input value={sharedValue} fun1={setSharedValue} label="Input 1" />

      <Input value={sharedValue} fun1={setSharedValue} label="Input 2" />

    </div> );
```

```
}

export default App;
```

Child Component (`Input`):

```
function Input({ value, fun1, label }) {
  return (
    <div>
      <label>{label}</label>
      <input
        type="text"
        value={value}
        onChange={(e) => fun1(e.target.value)}
      />
    </div>
  );
}
  export default Input ;
```

How It Works

1. **State in Parent**:
   - The `App` component holds the `sharedValue` state and the `setSharedValue` function.
2. **Passing Props**:
   - The `sharedValue` is passed as the `value` prop, and the `setSharedValue` function is passed as the `fun1` prop to both `Input` components .
3. **Updating State**:
   - When a user types into either input field, the `onChange` handler updates the `sharedValue` state in the parent.
   - Both input fields re-render with the updated value since they share the same state.

Common Use Cases for State Lifting

1. **Form Validation**:
    - Centralizing form data and validation logic in the parent component.
2. **Synchronizing Components**:
    - Keeping multiple components in sync, such as inputs, sliders, or toggles.
3. **Dynamic Data Display**:
    - Updating one component's state and reflecting the changes in related components (e.g., filtering a list and updating a chart).

Conditional Rendering :

LoginBtn.jsx :

```
const LoginBtn = () => {
  return (
    <div>
      <button>Login</button>
    </div>
  )
}
```

LogoutBtn.jsx :

```
const LogoutBtn = () => {
  return (
    <div>
      <button>Log Out</button>
    </div>
  )
}
```

App.js :

```
  const [isLogin,setLogin]  = useState(false);
```

1. if else

```
  if(isLogin==true)
  {
    return (
      <div>
      <LogoutBtn/>
      </div>
  )
}
  else
  {
    return (
      <div>
      <LoginBtn/>
      </div>
    )
  }
```

2. Ternary operator

```
return (
  <div>
  {isLogin ? <LogoutBtn/> : <LoginBtn/> }
  </div>
)
```

3. Bitwise Operator

```
return (
  <div>
    {isLogin && <LogoutBtn/>}
    {isLogin!=true && <LoginBtn/>}
  </div>
)
```

Event Handling in React :

Event handling in React allows developers to create interactive user interfaces by listening to and responding to user interactions such as clicks, key presses, form submissions, and more .

React provides a declarative and consistent way to handle events, which are similar to handling DOM events but with slight differences.

Common Events in React

Here are some common events used in React:

Event	Description
`onClick`	Fired when an element is clicked.
`onChange`	Fired when an input value changes.
`onSubmit`	Fired when a form is submitted.
`onMouseEnter`	Fired when the mouse enters an element.
`onKeyDown`	Fired when a key is pressed.

Basic Event Handling Example :

```
function App()
{
  function handleButtonClick()
  {
    alert('Button clicked');
  }
  function handleMouseOver()
  {
    alert('MouseOver on paragraph');
  }
  function handleInputChange(e)
  {
    console.log("Value till now " , e.target.value);
  }
  function handleSubmit(e)
  {
    e.preventDefault() ;    //It will prevent the default behaviour in form submission
    alert('Form submitted');
  }
  return (
    <div className="App">
      <div>
      <button onClick={handleButtonClick}>Click me (Button-1)</button>
      </div>

        <div>
    <button onClick={()=>{alert('Button clicked ')}}>Click me (Button-2)
       </button>
      </div>

      <div style={{border : "2px solid black" , padding:"10px" }}>
```

```
        <p onMouseOver={handleMouseOver}>I am paragraph and please hover me</p>
      </div>

      <div style={{padding:"10px" }}>
        <input type="text" onChange={handleInputChange} />
      </div>

      <div>
        <form onSubmit={handleSubmit}>
          <input type="text" /> <br></br> <br></br>
          <button type="submit">Submit</button>
        </form>
      </div>

    </div>
  );
}
```

NOTE : Avoid immediate invocation or function calling on event listener
Example :

In-line styling :

color : ‘red’, border : ‘2px solid black’ } } > I am a para </p>

React Routing :

Routing in React is typically handled using **React Router**, a popular library that allows you to navigate between different pages (or views) in a React application without reloading the page .

Installation

First, install `react-router-dom` package (for web applications)

—-> npm install react-router-dom

We use createBrowserRouter to create a router .

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

const router=createBrowserRouter(
  [
    {
      path : "/",
      element : <Home/>
    },
    {
      path : "/about",
      element : <About/>
    },
    {
      path : "/dashboard",
      element : <Dashboard/>
    }
  ]
);
```

Here we are providing the array of routes in the router we have created .Route contains the path and the element .

Now we have to inform browser about the router we have created using routerProvider

```
function App() {
  return (
    <div className="App">
      <RouterProvider router={router} />
    </div>
  );
}
```

OR

Use <BrowserRouter/> component

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

<BrowserRouter>
 <NavBar />
 <Routes>
<Route path="/" element={<Home />} />
<Route path="/about" element={<About />} />
 <Route path="/dashboard" element={<Dashboard />} />
 </Routes>
 </BrowserRouter>
```

Here <NavBar/> component will be always there in the page .

Now we have to create a navbar by creating a <Navbar/> Component

```
import React from 'react'
import { NavLink } from 'react-router-dom'
const Navbar = () => {
  return (
    <div>
      <ul>
        <li>
           <NavLink to='/'>Home</NavLink>
        </li>
        <li>
           <NavLink to='/about'>About</NavLink>
        </li>
        <li>
           <NavLink to='/dashboard'>Dashboard</NavLink>
         </li>

      </ul>
    </div>
```

```
  )
}
export default Navbar
```

NOTE : We don’t use <a> tag in React because it loads the whole page .

Instead we use <Link> tag or <NavLink> tag

<NavLink> tag provides an in-build class with the name ‘isActive’ using which we can highlight the current page name in the Navbar .

```
<ul>
        <li>
   <NavLink to='/'  className={({isActive})=>isActive ? "activeLink": "" }>
Home</NavLink>
        </li>
        <li>
  <NavLink to='/about' className={({isActive})=>isActive ? "activeLink": ""
}>About</NavLink>
        </li>
        <li>
           <NavLink to='/dashboard' className={({isActive})=>isActive ?
"activeLink": "" }>Dashboard</NavLink>
         </li>
      </ul>
```

Here we are adding a class with the name activeLink if the page is active .

Navigating using useNavigate() Hook in :

The `useNavigate()` hook in React Router is used for programmatic navigation within a React application. It replaces the older `useHistory()` hook from React Router v5.

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

const MyComponent = () => {
  const navigate = useNavigate();

  const goToHome = () => {
    navigate('/home');
  };

  return <button onClick={goToHome}>Go to Home</button>;
};
```

Nested Navigation :

```
const router=createBrowserRouter(
  [
     {
      path : "/dashboard",
      element :
      <div>
      <Navbar/>
      <Dashboard/>
   </div>,
   children:[
    {
      path:'courses',
      element:<Course/>
    },
    {
      path:'reports',
      element:<Reports/>
    },{
      path:'mocktest',
      element:<Mocktest/>
    }   ]    }  ]   );
```

```
function App() {
  return (
    <div className="App">
      <RouterProvider router={router}/>
    </div>
  );
}
```

In Dashboard.jsx

```
const Dashboard = () => {
  return (
    <div>
      DashBoard Page
      <nav>
        <Link to="courses">Course</Link> <br />
        <Link to="reports">Reports</Link>  <br />
        <Link to="mocktest">Mock Test</Link> <br />
      </nav>
      <Outlet/>    —-> Don’t forget to add <Outlet/>  tag
    </div>
  )
}
```

React-hook-form

React-hook-form is a lightweight, performant, and easy-to-use library for managing forms in React.

1. Installation :

npm install react-hook-form

2. Basic Form Example

```
import './App.css';
import { useForm } from 'react-hook-form';
function App() {
  const {
     register,
     handleSubmit,
     watch,
      formState: { errors }
     } = useForm();
      function onSubmit(data)
      {
        console.log('Form submitted successfully',data);
      }
  return (
    <div className="App">
      <form onSubmit={handleSubmit(onSubmit)}>
        <div>
          <label>FirstName :</label>
          <input {...register('FirstName :')}/>
        </div> <br/>
        <div>
          <label>MiddleName :</label>
          <input {...register('MiddleName :')}/>
        </div> <br/>
        <div>
          <label>LastName :</label>
          <input {...register('LastName :')}/>
        </div> <br/>
        <div>
          <input type="submit"/>
        </div>
      </form>
```

```
  </div>
 );
}
export default App;
```

3. Adding validations

```
<input {...register('FirstName :' ,
        {required:true,
        minLength:{value:3, message:'MinLength must be greater than 3'},
        maxLength:10
        })}/>
```

Redux Toolkit - Advanced topic

Read this below documentation once

https://react-redux.js.org/tutorials/quick-start

OR

Watch this lecture once

https://www.youtube.com/watch?v=DnRY5yG67u8&list=PLDzeHZWIZsTo0wSBcg4-NMIbC0L8evLrD&index=72

useRef Hook in React :

Use 1 : The `useRef` hook in React is used to persist values across renders without causing a re-render when its value changes .

Syntax :

```
import { useRef } from "react";
```

const refContainer = useRef(initialValue);

useRef Hook returns an object and the value of the variable will be stored in the key named current .

Example :

Import {useRef} form “react”;
const cnt = useRef(12);
console.log(cnt.current);
cnt.current =cnt.current + 1;
console.log(cnt.current);

NOTE : We were using useState Hook before this . But the useState hook re-renders the whole component every time the value of useState variable changes .
Re-rendering the component re-initializes all the variables defined using let , var etc .
Example :

```
import './App.css';
import { useState } from 'react';
import { useEffect } from 'react';

function App() {
const [count,setCount] = useState(0);
let x=10;
useEffect(() => {
   console.log('re-rendering the whole component');
 })

function handleClick()
{
   x=x+1;
   console.log('The value of x :',x);
   setCount(count+1);
 }
```

```
    return (
      <div className="App">
         <button>{count}</button> <br/> <br/>
        <button onClick={handleClick}>+</button>
      </div>
    );
  }

  export default App;
```

Here useState Hook re-renders the whole <APP> component as soon as the value of count variable changes i.e when onClick() event triggers .And as soon as useState re-renders the whole component the value of x initializes to its initial value .
That's why we use useRef Hook in react .

In short the variables created using useRef persist their value across re-renders .

Use 2 : It is commonly used for accessing and interacting with DOM elements directly . (v.v IMP point)

OR

for storing mutable values that don't trigger re-renders.

Step 1 : Create the reference
Step 2 : Link the reference
Step 3 : Manipulate dom using the reference

Example :

```
    import { useRef } from 'react';
    import './App.css';

    // Use Case 2 : It is commonly used for accessing and interacting with DOM
elements directly .
```

```
   function App() {
   const btnRef=useRef();  // Create the reference
   function handleClick()
   {
btnRef.current.style.backgroundColor="red"; //Manipulate DOM using reference
   }
   return (
       <div className="App">
         <button ref={btnRef} onClick={handleClick}>Change My color</button>
             // Link the reference
       </div>
   );
   }

   export default App;
```

In React, `target.value` is often used in `onChange` handlers to manage form state .

```
import { useState } from "react" ;
function App() {
   const [text, setText] = useState("") ;

   const handleChange = (event) => {
      setText(event.target.value);
   };
   return (
      <div>
         <input type="text" value={text} onChange={handleChange} />
         <p>Typed: {text}</p>
      </div>
   );
}
export default App;
```

Passing parameterized function in onClick() event :

In React, when using an `onClick` event listener with a parameter, you need to **pass a function reference** rather than calling the function directly. Here's how you can do it :

Using an Arrow Function (Recommended)

```
function App() {
    const handleClick = (name) => {
        alert(`Hello, ${name}!`);
    };
    return (
        <button onClick={() => handleClick("John")}>Click Me</button>
    );
}
export default App;
```

The arrow function `() => handleClick("John")` ensures `handleClick` only runs when clicked.

Mapping in React :

In React, you can use the `.map()` function to iterate over an array and dynamically render elements. Here's how :

1️⃣ Basic Example: Mapping a Simple Array

```
function App()
{
   const fruits = ["Apple", "Banana", "Mango", "Orange"];
   return (
      <div>
         <h2>Fruits List</h2>
         <ul>
            {fruits.map((fruit, index) => (
               <li key={index}>{fruit}</li>
            ))}
         </ul>
      </div>
   );
}
export default App;
```

✔ **Explanation:**

- `.map()` loops through the `fruits` array.
- Each item is wrapped inside an `<li>` element.
- The `key` prop ensures React efficiently updates the list.

2️⃣ Mapping an Array of Objects

If your array contains objects, you can map through it like this:

```
function App() {
    const users = [
        { id: 1, name: "John", age: 25 },
        { id: 2, name: "Jane", age: 30 },
        { id: 3, name: "Alice", age: 22 }
    ];
    return (
        <div>
            <h2>User List</h2>
            <ul>
                {users.map((user) => (
                    <li key={user.id}>
                        {user.name} - Age: {user.age}
                    </li>
                ))}
            </ul>
        </div>
    );
}
export default App;
```

✔ Key Points:

- Always use a **unique key** (e.g., `user.id`) to optimize rendering.
- Access object properties using `user.name`, `user.age`, etc.

3️⃣ Mapping an Array to a Table

```
function App() {
    const products = [
        { id: 101, name: "Laptop", price: 1200 },
        { id: 102, name: "Phone", price: 800 },
        { id: 103, name: "Tablet", price: 500 }
 ];
    return (
```

```
        <div>
            <h2>Product List</h2>
            <table border="1">
                <thead>
                    <tr>
                        <th>ID</th>
                        <th>Name</th>
                        <th>Price</th>
                    </tr>
                </thead>
                <tbody>
                    {products.map((product) => (
                        <tr key={product.id}>
                            <td>{product.id}</td>
                            <td>{product.name}</td>
                            <td>${product.price}</td>
                        </tr>
                    ))}
                </tbody>
            </table>
        </div>
    );
}
export default App;
```

✔ Best Practices:

- Use `<table>` for structured data.
- `.map()` generates rows dynamically.
- Unique `key={product.id}` prevents unnecessary re-renders.

You can pass a **JavaScript object** to the `style` attribute.

```
function App()
{
    const boxStyle = {
        width: "200px",
        height: "200px",
        backgroundColor: "lightblue",
        borderRadius: "10px",
        textAlign: "center",
        lineHeight: "200px",
    };

    return <div style={boxStyle}>Styled Box</div>;
}
export default App;
```

✔ **Why ?**

- Styles are directly defined as a JavaScript object.
- No need for an external CSS file.
- Uses **camelCase** instead of hyphenated CSS properties (e.g., `backgroundColor` instead of `background-color`).

How to toast a pop-up message on onClick () :

Step 1: Install `react-hot-toast library`

`react-hot-toast` is a lightweight and customizable toast notification library for React.

Step 2: Import toast and Toaster into the current file

```
import { Toaster, toast } from 'react-hot-toast';
```

Step 3: Add <Toaster /> component in the current file

<Toaster /> {/* Required to render toasts */}

Step 4: Add toast message in the onClick function

toast.success("Product Added Successfully !");

Types of Toasts :

toast.success('Success message! ✅');
toast.error('Error message! ❌');
toast.loading('Loading... ⏳');
toast('Simple message');

How to add toggling feature :

```
const [isCheck,SetIsCheck] =  useState(false);

 function toggleCheck()
 {
   SetIsCheck(!isCheck);
 }

 <button  onClick={toggleCheck}>{isCheck.toString()}</button>
```

NOTE : Here we are using the toString() method to convert boolean value into string because boolean value can’t be printed like that .

Managing State with an Array in React :

NOTE : Component will re-render only if the value of state changes . But in case of arrays and objects even if we change the value reference still remains the same . So while updating the value use spread operator . Because the spread operator passes the copy of an array or object which has different references .

In React, managing state with an **array** is common, especially when dealing with lists of items like tasks, users, or products. However, since React state should be **immutable**, we should always update arrays in a way that creates a new array instead of modifying the existing one .

1️⃣ Using `useState` with an Array

The `useState` hook allows us to store and update an array in a functional component.

Example: Managing a List of Items

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

function ItemList() {
  const [items, setItems] = useState(["Apple", "Banana", "Orange"]);
  return (
    <div>
      <h2>Fruits</h2>
      <ul>
        {items.map((item, index) => (
          <li key={index}>{item}</li>
        ))}
      </ul>
    </div>
  );
}
```

```
export default ItemList;
```

🔹 The `useState` hook initializes `items` as an **array** and renders it using `map()`.
🔹 However, the list is **static**—let's learn how to **add, update, and remove** items dynamically.

2️⃣ Adding Items to an Array in State

Since React state is **immutable**, we should not use `push()`. Instead, we create a new array using the **spread operator (`...`)**.

Example: Adding an Item to an Array

```
function ItemList() {
  const [items, setItems] = useState(["Apple", "Banana", "Orange"]);

  const addItem = () => {
    setItems([...items, "Mango"]);  // Creates a new array with the new item
  };

  return (
    <div>
      <h2>Fruits</h2>
      <ul>
        {items.map((item, index) => (
          <li key={index}>{item}</li>
        ))}
      </ul>
      <button onClick={addItem}>Add Mango</button>
    </div>
  );
}
```

🔷 Why use `[...items, " Mango "]` ?

- It creates a new array, keeping React's **immutability principle**.
- `push()` would modify the existing array instead of creating a new one, which React doesn't detect properly .

3️⃣ Removing (delete) an Item from an Array

To remove an item, we use **`filter()`** to create a new array **without modifying the original one**.

Example: Removing an Item by Name

```
function ItemList() {
  const [items, setItems] = useState(["Apple", "Banana", "Orange"]);
  const removeItem = (itemToRemove) => {
      setItems(items.filter((item) => item !== itemToRemove)); // Creates a new filtered array
  };
  return (
    <div>
      <h2>Fruits</h2>
      <ul>
        {items.map((item, index) => (
          <li key={index}>
            {item} <button onClick={() => removeItem(item)}>Remove</button>
          </li>
        ))}
      </ul>
    </div>
  );
}
```

NOTE : Avoid pop,shift, splice to remove an item , just use filter or slice .

4️⃣ Updating (replace) an Item in an Array

To update an item, use `map()` to create a new array with the modified value.

Example: Updating an Item

```
function ItemList() {
  const [items, setItems] = useState(["Apple", "Banana", "Orange"]);

  const updateItem = (oldItem, newItem) => {
    setItems(items.map((item) => (item === oldItem ? newItem : item)));
          // Replaces oldItem with newItem
  };

  return (
    <div>
      <h2>Fruits</h2>
      <ul>
        {items.map((item, index) => (
          <li key={index}>
            {item}{" "}
                <button onClick={() => updateItem(item, "Grapes")}>Change to
Grapes</button>
          </li>
        ))}
      </ul>
    </div>
  );
}
```

Why use `map()` ?

- It creates a new array, replacing only the specified item.

5️⃣ Managing Arrays of Objects in State

If your array contains **objects**, you need to **preserve the existing properties** while updating.

Example: Updating an Array of Objects

```
function UsersList() {
  const [users, setUsers] = useState([
    { id: 1, name: "Alice", age: 25 },
    { id: 2, name: "Bob", age: 30 },
  ]);
  const increaseAge = (id) => {
    setUsers(users.map((user) =>
      user.id === id ? { ...user, age: user.age + 1 } : user
    ));
  };
  return (
    <div>
      <h2>Users</h2>
      <ul>
        {users.map((user) => (
          <li key={user.id}>
            {user.name} - Age: {user.age}{" "}
```

```
        <button onClick={() => increaseAge(user.id)}>Increase Age</button>
         </li>
       ))}
     </ul>
   </div>
  );
}
```

🔷 **Why use `{ ...user, age: user.age + 1 }`?**

- This ensures that the **entire object is updated immutably** without modifying other properties.

NOTE : READ THE DOCUMENTATION :

—-----><u>https://react.dev/learn/updating-arrays-in-state</u>

Managing state with an array in React requires immutable updates using array methods like **`map()`**, **`filter()`**, and **`spread`** **`(...)`**. Whether you are **adding, removing, or updating** items, always create a new array to ensure React detects the change and re-renders the component correctly.

Managing State with an Object in React :

When managing state in React with an **object**, you need to ensure that you correctly update the state without **mutating** the original object. React state updates should always be **immutable**, meaning you should create a new object instead of modifying the existing one.

Using `useState` with an Object :

```
import React, { useState } from "react";
function UserProfile() {
  // Initializing state with an object
  const [user, setUser] = useState({
    name: "Alice",
    age: 25,
    location: "New York"
  });

  // Function to update the age
  const increaseAge = () => {
    setUser((prevUser) => ({
      ...prevUser,  // Spread operator to keep other properties unchanged
      age: prevUser.age + 1  // Updating only the age
    }));
  };
  return (
    <div>
      <h2>{user.name}</h2>
      <p>Age: {user.age}</p>
      <p>Location: {user.location}</p>
      <button onClick={increaseAge}>Increase Age</button>
    </div>
  );
```

}

export default UserProfile;

🔷 **Why use `...prevUser`?**

- This ensures **other properties (name, location) remain unchanged** while updating the `age` property.
- If you don't use `...prevUser`, React will **replace the entire state object**, losing the other properties.

NOTE : READ THE DOCUMENTATION :

—-----><https://react.dev/learn/updating-objects-in-state>

Logical and presentational components in react :

Logical vs Presentational Components in React

In **React**, components are often categorized into two types:
1️⃣ **Presentational Components** (UI-focused)
2️⃣**Logical Components** (a.k.a. Container Components, state management-focused)

This pattern helps keep your code **clean**, **modular**, and **maintainable**.

1️⃣ Presentational Components (UI Components)

Definition:

- Focus **only on rendering UI** (HTML & CSS).
- Receive **props** and display data.
- **Do NOT manage state** or handle complex logic.
- Can be **functional components** (mostly) or class-based .

Example of a Presentational Component

```
function UserCard({ name, age }) {
  return (
    <div className="card">
      <h2>{name}</h2>
      <p>Age: {age}</p>
    </div>
  );
}
export default UserCard;
```

🔷 Key Characteristics:

✔ **Receives props** and displays data.
✔ **Stateless** (doesn't use `useState`).
✔ **Reusable** in multiple places.

2️⃣ Logical (Container) Components

Definition:

- Handle **state, API calls, and business logic**.
- **Pass data** as props to presentational components.
- Usually **functional components** use hooks (`useState`, `useEffect`).

Example of a Logical Component

```
import React, { useState, useEffect } from "react";
import UserCard from "./UserCard"; // Importing presentational component

function UserContainer () {
  const [user, setUser] = useState({ name: "", age: 0 });

  useEffect(() => {
    // Simulate API call
```

```
    setTimeout(() => {
      setUser({ name: "Alice", age: 25 });
    }, 1000);
  }, []);

   return <UserCard name={user.name} age={user.age} />; // Passing data as props
}
export default UserContainer;

import React, { useState, useEffect } from "react";
import UserCard from "./UserCard"; // Importing presentational component

function UserContainer() {
  const [user, setUser] = useState({ name: "", age: 0 });

  useEffect(() => {
    // Simulate API call
    setTimeout(() => {
      setUser({ name: "Alice", age: 25 });
    }, 1000);
  }, []);

   return <UserCard name={user.name} age={user.age} />; // Passing data as props
}

export default UserContainer;
```

🔷 Key Characteristics:

✔ **Manages state & side effects** (`useState`, `useEffect`).
✔ **Fetches data** (API calls, event handling).
✔ **Passes props** to presentational components.

Simple Form :

```
import { useState } from 'react'
import './App.css'
function App()
{
  const [data, setData] = useState({
    FirstName : "",
    MiddleName : "",
    LastName : "",
    PhoneNumber : "",
    Gender : "",
    About : "",
  })
  function handleChange(event)
  {
    event.preventDefault();
    setData({...data,
      [event.target.name]:[event.target.value]}
    )
  }
  function showData(event)
  {
    event.preventDefault();
    console.log(data);
  }
  return (
    <>
    <main>
      <div>
      <form action="" onSubmit={showData}>
        <h2>Simple Form</h2>
        <label htmlFor="firstname">FirstName : </label>   <br />
            <input type="text" id='firstname' name='FirstName' placeholder='Enter First Name :'
onChange={handleChange}/> <br /> <br />
        <label htmlFor="secondname">MiddleName : </label>   <br />
          <input type="text" id='secondname' name='MiddleName' placeholder='Enter Middle Name
:' onChange={handleChange}/> <br /> <br />
        <label htmlFor="lastname">LastName :</label>   <br />
            <input type="text" id='lastname' name='LastName' placeholder='Enter Last Name :'
onChange={handleChange}/> <br /> <br />
        <label htmlFor="phoneNo">Phone No :</label>   <br />
          <input type="number" id='phoneNo' name='PhoneNumber' placeholder='Enter Phone
Number :' onChange={handleChange}/> <br />
```

```
        <h3>Gender</h3>
        <div className="gender">
        <div>
        <label htmlFor="male">Male</label>
        <input type="radio" id='male' name='Gender' value='male' onChange={handleChange}/>
        </div>
        <div>
        <label htmlFor="female">Female</label>
                                <input type="radio" id='female' name='Gender' value='female'
onChange={handleChange}/>
        </div>
        <div>
        <label htmlFor="other">Other</label>
        <input type="radio" id='other' name='Gender' value='other' onChange={handleChange}/>
        </div><br /> <br />
        </div>
        <label htmlFor="about">About : </label> <br />
        <textarea name="About" id="about" onChange={handleChange}></textarea> <br /> <br />
        <div id='resetAndSubmit'>
        <input type="reset" />
        <input type="submit" />
        </div>
      </form>
      </div>
      <div className='hide'>
        <h1>Data :</h1>
        <li>FirstName : {data.FirstName}</li>
        <li>MiddleName : {data.MiddleName}</li>
        <li>LastName : {data.LastName}</li>
        <li>Phone Number : {data.PhoneNumber} </li>
        <li>Gender : {data.Gender}</li>
        <li>About : {data.About}</li>
      </div>
      </main>
    </>
  )
}
export default App
```

How to deploy react apps on netlify :

Step 1 : Run the following command in the terminal of react app folder

```
npm run build
```

This will create a folder named ‘dist’ which includes all the necessary files along with index.html

Step 2 : Now One can upload a ‘dist’ folder and deploy their react apps on netlify.

Step 3 : And if we use react router in the react project , then create a file naming netlify.toml and put the following code in that file .

```
[[redirects]]
from="/*"
to="/index.html"
status=20
```