REACT JS

- **What is React ?**
  - → open source library for building user interfaces
  - → Not a framework
  - → Focus on UI
  - → Rich ecosystem
  - → Developed at Facebook in 2011
  - → used to create single page application and for building UIs.

- **key features**
  - ↳ Component based Architecture : Encapsulate UI element into components
  - ↳ Virtual Dom : Efficiently updates and renders components by comparing new elements with previous version.
  - ↳ JSX (Javascript XML) : syntax extension that allows mixing HTML with Javascript.
  - ↳ One-way Data Binding : Unidirectional data flow simplifies the understanding of how data changes impact the UI.
  - ↳ React Hooks : Introduce state and lifecycle features to functional components.

- **Setting up React**

  Step 1 : Install Node.js and npm

  ① Download the LTS version (Long-term-support)

  ② Verify Installation :

  node -v (check Node.js version)

  npm -v (check npm version)

npm → node package manager
npx → node package executor

Step 2: Install Create React App
① npm install -g create-react-app

Step 3: Create a New React Project
① create project
npx create-react-app my-app
↳ project name
② Navigate into project directory
cd my-app

Step 4: Start the development Server
npm start or npm run start

Above steps are time consuming. Therefore we use Vite

step ① npm create vite@latest
Project name: 01vitereact
select a Framework:
Vanilla
Vue
React ✓
:
other
Select a Variant:
Typescript
Type script + SWC
Javascript ✓
Javascript + SWC

step ② cd 01vitereact
dir
npm install
npm run dev

React Dom → for web
React Native → for mobile app

PAGE NO.:
DATE: / /

- **Project Structure**

1. node_module/ : has all installed node packages
2. public/ : contains static files that don't change.
3. src/ : main folder for the React code.
   (i) components/ : Reusable part of the UI, like buttons or headers.
   (ii) assets/ : Images, fonts, and other static files.
   (iii) styles/ : CSS or Stylesheets.
4. package.json contain information about this project like name, version, dependencies on other react packages.
5. vite.config.js contains vite config

Example 1: src/App.jsx/main.jsx

App.jsx

```
function App() {
    return (
        <h1> Hii Kamlesh here </h1>
    )
}
export default App;
```

main.jsx

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

Example 2: src/ App.jsx / main.jsx / practice.jsx

practice.jsx — function name → capital

```
function Practice() {
    return (
        <h3> This is my first react Project </h3>
    )
}
export default Practice;
```

App.jsx

```
import Practice from "./practice"
function App() {
    return (
        <Practice />
        <h1> practice test </h1>
        <p> test para </p>
    )
}
export default App
```

Error because we can only return one element we can solve it by using or fragment <>.

Note: Fragment

```
<>
...
</>
```

```
import Practice from "./practice"
function App() {
    return (
        <>
        <Practice />
        <h1> practice test </h1>
        <p> test para </p>
        </>
    )
}
export default App;
```

main.jsx file same as example1

Pros and Cons of React JS:

① Advantages:
- Easy to learn and use
- creating of dynamic web application become easier
- Reusable components
- Performance enhancement.
- The support of Handy Tools
- Known to be SEO friendly
- The Benefits of having JS library.

② Disadvantages
- The high pace of development
- ~~Poor~~ React JS only cover UI layer of the app nothing else
- JSX is barrier.

React JSX

→ JSX provide you to write HTML/XML like ~~code~~ structure in the same file where you write Javascript code, then preprocessor will transform these expression into actual Javascript code. Just like XML/HTML, JSX tags name, attribute, and children.

- ## Why use JSX?

→ It is faster then regular Javascript beoz it perform optimization while translating the code to javascript

→ Insted of seperating technologies by putting markup

and logic in seperate files, React uses component that contain both, we will learn component
→ It is type-safe, and most of the errors can be found at compilation time
→ It make easier to create template.

React Components

There are two types of components
(1.) Class Based Components
(2.) Function Based Components

- Class Based components
→ Before making class based components we need to inherit function from React.Component and this can be done with extends.
Eg.

```
class Dog extends React.Component{
    render(){
        return <h1>Bark</h1>;
    }
}
```

→ It also requires a render method which return HTML
→ Not Preferred anymore

- Function Based Components
→ In function it's simpler, we just need to return the HTML

Eg.

```
function Dog() {
    return <h1> Bark </h1>
}
```

→ More Popular.

NOTE: Component's name must start with uppercase letter.

- Rendering a Component
  → we made a component, now we want to render / use it.
  → Syntax:

```
<ComponentName />
```

- Component in files
  → To have less mess inside main file (with all the components in the same file) and to reuse components on different pages, we have to make them separately. So that we can just import them in any files and use them!
  → For that we just make a new file called Dog.js, make class or function based component there and export default that class/function.

Eg.

Dog.js

```
function Dog() {
    return <h1> Bark </h1>;
}
export default Dog;
```

Note: file name must start with uppercase letter.

PAGE NO.
DATE / /

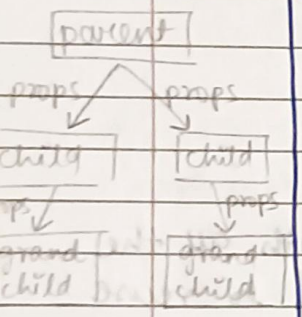
Props → are the arguments passed to React component via HTML attributes.

→ In React "props" (short for properties) are a mechanism for passing data from one component to another component, typically from parent to a child component. Props allows you to configure a component with data that can influence its rendering and behaviour

Eg. Parent Component.jsx

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

const ParentComponent = () => {
    return (
        <div>
            <ChildComponent name="John" />
        </div>
    );
};
export default ParentComponent;
```



Key points:
- Data flows one-way (downwards)
- Props are immutable
- Used for communication b/t components

Usage:
- Pass Data from parent to child component
- Make components reusable
- Defined as attribute in JSX

Child Component.jsx

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

const ChildComponent = (props) => {
    return <h1> Hello, {props.name}! </h1>
};

export default ChildComponent;
```

In this example 'Parent component' passes a prop called 'name' with the value 'John' to 'ChildComponent', which then render "Hello, John!"

→ As mentioned earlier we can import the same component in different files and use it, but maybe in different files some changes in the component is needed. For that, we can use props!

Component :-

Eg.
```
function Dog (props) {
    return <h1> Dog color is {props.color}
        </h1>
}
```

mainfile :

```
<Dog color = "black" />
```

React Fragment

→ `React.fragment` is a component that allow you to group multiple elements without adding extra node to the Dom.

→ This is useful when you want to return multiple element from a component's render method but don't want to wrap in a div or any other container element.

Eg.
```
import React from 'react';
const Demo = () => {
    return (
        <div>
            <h1> Title </h1>
            <p> Paragraph </p>
        </div>
    );
};
export default Demo;
```

OR

```
import React from 'react';
const Demo = () => {
    return (
        <React.Fragment>
            <h1> Title </h1>
            <p> Paragraph </p>
        </React.Fragment>
    );
};
export default Demo;
```

Short syntax :-

→ using empty tag (`<>` and `</>`), which make the code cleaner.

~~Map Method~~

Eg.

App.jsx

```
import React from "react";
import "bootstrap/dist/css/bootstrap.min.css";
import "./App.css";
function App() {
  <>
    <h1> Healthy Food </h1>
    <ul class = "list-group">
      <li class = "list-group-item"> Dal </li>
      <li class = "list-group-item"> Salad </li>
      <li class = "list-group-item"> Roti </li>
    </ul>
  </>
}
export default App;
```

O/p:

Healthy Food
- Dal
- Salad
- Roti

- **Map Method**

→ Purpose : Render lists from array data

→ JSX Elements : Transform array items into JSX

→ Inline Rendering : Directly inside JSX

{items.map(item => <li key={item.id}>{item.name})}

→ Key Props : Assign unique key for optimized re-renders.

{item.name}

Eg. import React from 'react';

```
const Namelist = () => {
    const names = ['Alice', 'Kamlesh', 'Mahim'];
    return (
        <ul>
            { names.map (name => <li class="list-group">
                                    {name}</li>)}
        </ul>
    );
};
export default Namelist;
```

Eg. with more complex Data

```
import React from 'react';
const UserList = () => {
    const users = [
        { id: 1, name: 'Alice', age : 25},
        { id: 2, name: 'Kamlesh', age : 21},
        { id: 3, name: 'Mahim', aage: 21},
    ];
    return (
        <ul>
            { users.map(user) => (
                <li key = {user.id}>
                    {user.name} is {user.age} year old.
                </li>
            ))}
        </ul>
    );
};
export default UserList;
```

Conditional Rendering

→ Displaying content based on certain conditions.
→ Allows for dynamic user interfaces.

Methods :

⊙ If-else statement : Choose b/t two blocks of content

Eg.
```
function App() {
    let FoodItem = [];
    if (foodItems.length === 0) {
        return <h3> I am still hungry. </h3>;
    }
    return (
        <>
        <h1> Healthy Food </h1>
        <ul className = "list-group">
            { foodItems.map ((item) => (
                <li key={item} className="list-group-item"> {item} </li>
            ))}
        </ul>
    </>
    );
}
export default App;
```

⊙ Ternary operators : Quick way to choose between two options

Eg.
```
function App() {
    let foodItem = [];
    return (
        <>
            <h1> Healthy food </h1>
            ; {foodItems.length === 0 ? <h3> I am still
                                    hungry. </h3> : null }
            <ul className = "list-group">
                {foodItems.map ((item) => (
                    <li key = {item} className = "list-group-
                                    -item"> {item}
                    </li>
                ))}
            </ul>
        </>
    );
}
export default App;
```

Note:

'let empty msg =

⊙ Logical operator : useful for rendering content when a condition is true.

Eg.
```
function App () {
    let foodItems = [];
    return (
        <>
            <h1> Healthy food </h1>
            {foodItems.length === 0 && <h3> I am still hungry
                                    </h3>}
            <ul>
                ....
            </ul>
        </>
    );
}
export default App;
```

* Important *

- using Props in the same example :

Components

FoodItems.jsx

```
import Item from "./Item";
const FoodItems = ({items}) =>
    => {
  return (
    <ul className="list-group">
      { items.map ((item)
        => (<Item key={item}
          FoodItem = {item}
        ></Item>
      ))}
    </ul>
  );
};
export default FoodItems;
```

ErrorMessage.jsx

```
const ErrorMsg = ({items}) => {
  return
    <>
    {items.length === 0 &&
      <h3> I am still
      hungry</h3>}
    </>
};
export default ErrorMsg;
```

Item.jsx

```
const Item = ({FoodItem})
    => {
  return
    <li className =
      "list-group-item">
      {FoodItem}
    </li>;
};
export default Item;
```

App.jsx

```
import FoodItems from "./components/FoodItems";
Import ~~ErrorMessage~~ ErrorMsg from "./components/ErrorMessage";
import "./App.css";
import "bootstrap/dist/css/bootstrap.min.css";
function App() {
   let foodItem = ["Dal", "Green Vegies", "Roti", "Milk", "Ghee"];
   return (
     <>
     <h1> Healthy Food </h1>
     <ErrorMsg items = {foodItems}></ErrorMsg>
     <FoodItems items = {foodItems}></FoodItems>
   </>
  );
}
export default App;
```

PAGE NO.
DATE : / /

~~CSS Modules~~

React CSS Styling

→ There are three ways to style in React :

1. Inline Styling
2. CSS Stylesheets
3. CSS Modules.

① Inline Styling

→ To inline style an element, we can make a Javascript Objects.

```
const App = () => {
  return (
    <>
      <h1 style={{ backgroundColor:"black"}}>
        CodeWithKamCodes</h1>
    </>
  );
}
```

→ In the first curly brace is to write Javascript and second is to make a javascript object. We can also write it like :

```
const App = () => {
  const h1Style {
    backgroundColor : 'black';
  }
  return (
    <>
      <h1 style = {h1Style}> CodeWithKamCodes
      </h1>
    </>
  );
}
```

Note: CSS property name must be camelCase.
background-color would be backgroundColor

② CSS Stylesheets

→ You can save the whole CSS in a separate file with file extension.css and import it in your application.

App.css

```
body {
    backgroundColor : 'purple';
    color : 'white';
}
```

Here we are writing CSS, so we don't need to make JS object or do it in camelCase.

Index.js

```
import './App.css';
```

③ CSS Modules

→ In this you don't have to worry about name conflict as it is component specific. The CSS is available only for the file in which it is imported.

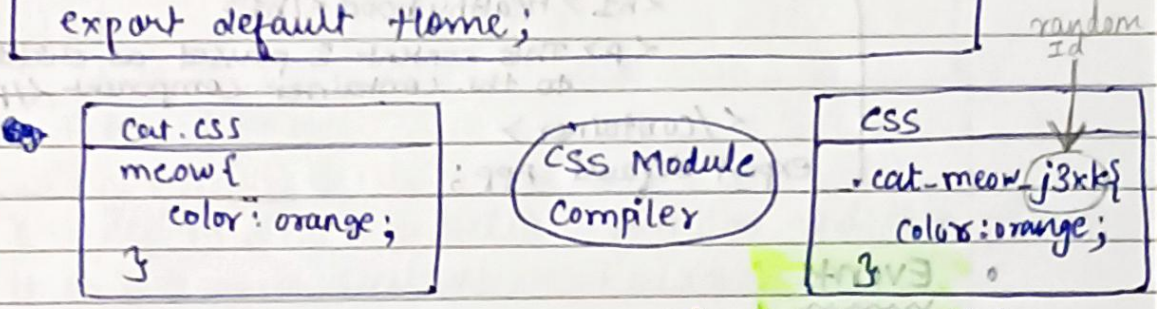
→ Make a file with extension .module.css

Eg. index.module.css

```
.button {
    background-color : 'purple';
    color : 'white';
}
```

Import it in component

```
import styles from './index.module.css';
const Home = () => {
    return (
        <button className={styles.button}>
            Click me! </button>
    )
};
export default Home;
```



1.) Localized class names to avoid global conflicts
2.) Styles are scoped to individual components
3.) Help in creating components - specific styles
4.) Automatically generates unique class names
5.) Promotes modular and maintainable CSS.
6.) Can use alongside global css when needed.

Passing Children

→ children is a special prop for passing element into components
→ Used for flexible and reusable component design
→ Common in layout or container components
→ Accessed with props.children
→ Can be any content : strings, numbers, JSX, or components.
→ Enhance component composability and reusability.

Container.jsx

Eg.

```
function Container =(props) =>{
    return <div>{props.children}</div>;
};
export default container;
```

App.jsx

```
function App(){
  return(
    <container>
      <h1> Healthy food </h1>
      <p> This content is passed as children
          to the container component </p>
    </container>
  export default App;
```

- **Events**

→ Just like HTML DOM events, React can perform actions based on user events. React has the same events as HTML : click, mouseover, change etc.

Adding an Event

↳ React events are written in camelCase syntax:
onClick instead of onclick

↳ React event handler are written inside curly braces:
onClick={shoot} instead of onclick="shoot()".

Eg. HTML :

```
<button onclick="shoot()">Take the Shot!</button>
```

REACT :

```
<button onClick={shoot}>Take the Shot!</button>
```

App.jsx

Eg.

```
export default function Button() {
    function handleClick() {
        alert('You clicked me!');
    }
    return (
        <button onClick={handleClick}>
            Click me
        </button>
    );
}
```

O/P: Click me

→ You defined the handleClick function and then passed it as a prop to <button>.

→ handleClick is an event handler

→ Event Handler functions :
- are usually defined inside your components
- Have names that start with handle, followed by the name of the event.

→ Alternatively, you can define an event handler inline in the JSX :

```
<button onClick={function handleClick() {
        alert('You clicked me!');
}}>
```

Or more concisely, using a arrow function

```
<button onClick={() => {
        alert('You clicked me!');
}}>
```

Passing Arguments

→ To pass argument to event handler, use an arrow function

Eg.
```
function football(){
    const shoot = (a) => {
        alert(a);
    }
    return (
        <button onClick={() => shoot("Goal")}>
            Take the shot!
        </button>
    );
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Football/>);
```

~~Passing Event Handler as props~~

Reading Props in Event Handler

Eg.
```
function AlertButton({message, children}){
    return (
        <button onClick={() => alert(message)}>
            {children}
        </button>
    );
}
export default function Toolbar(){
    return (
        <div>
            <AlertButton message="playing">
                Play Movie
            </AlertButton>
        </div>
    );
}
```

Passing event handler as props

Eg:

```
function Button ({onClick, children}) {
    return (
        <button onClick={onClick}>
            {children}
        </button>
    );
}
function PlayButton ({movieName}) {
    function handlePlayClick() {
        alert (`Playing ${movieName}!`);
    }
    return (
        <Button onClick={handlePlayClick}>
            Play "{movieName}"
        </Button>
    );
}
function UploadButton(){
    return (
        <Button onClick={() => alert ('Uploading')}>
            Upload Image
        </Button>
    );
}
export default function ToolBar(){
    return (
        <div>
            <PlayButton movieName="KGF" />
            <UploadButton />
        </div>
    );
}
```

O/P:

Play "KGF" | Upload Image

→ Here, ToolBar component render a PlayButton and an UploadButton :
- PlayButton passes handlePlayClick as the onClick prop to the Button inside
- UploadButton passes () ⇒ alert ('Uploading') as the onClick to the Button inside.

→ Finally your Button component accept a prop called onClick.

Naming event handler props

→ However, when you are building your own components, you can name their event handler props any way that you like!

→ By convention, eventhandler props should start with on, followed by capital letter.

Eg.
```
function Button ({onSmash, children}) {
    return (
        <button onClick={onSmash}>
            {children}
        </button>
    );
}

export default function App() {
    return (
        <Button onSmash={() ⇒ alert ('Playing')}>
            Play movie
        </Button>
    );
}
```

PAGE NO.:
DATE: / /

Preventing default Behaviour:

→ Some browser event have default behaviours associated with them. For example a <form> submit event, which happens when a button inside of it is clicked, will reload the whole page by default.

Eg. export default function signUp(){
    return(
        <form onSubmit={() => alert('submitted')}>
            <input />
            <button> send </button>
        </form>
    );
}

o/p: [ ] [send]

→ You can call e.preventDefault() on the event object to stop this form happening:

Eg. export default function signUp() {
    return (
        <form onSubmit={(e) => {
            e.preventDefault();
            alert('Submitting!');
        }}>
            <input />
            <button> send </button>
        </form>
    );
}

NOTE: e.preventDefault(): prevents the default browser for the few events that have it.

Event Propogation

→ Event propogation in React, like in standard HTML and JS, involve three phases : the capturing phase, the target phase and the bubbling phase. React primarily deals with the bubbling phase but can also handle event capturing.

Event Propogation Phases

① Capturing Phase : The event starts from the root of the document and travels down to the target element.

② Target Phase : The event reaches the target element.

③ Bubbling Phase : The event bubbles up from the target element back to the root of the document.

→ Eg. The contains two buttons. Both the and each button have their own onClick handlers. Which handler do you think will fire when you click on a button?

```
Eg. export default function Toolbar() {
      return(
        <div className="ToolBar". onClick={() => {
          alert('You clicked on the toolbar!');
        }}>
          </button onClick={() => alert('Playing')}>
            Play Movie
          </button>
          <button onClick={() => alert('Uploading')}>
            Upload Image
          </button>
      ); </div>
}
```

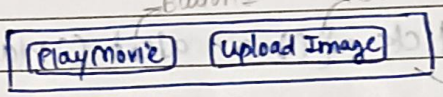
Play movie
2 alerts: Playing (OK) | You clicked on toolbar (OK)

Upload Images
2 alerts: Uploading (OK) | You clicked on toolbar (OK)

Toolbar + alert
You click on toolbar (OK)

PAGE NO.:
DATE: / /

O/P: Button 1 — Button 2 — Toolbar



→ If you click on either button, its onClick will run first, followed by the parent 's onClick. So two message will appear. If you click the toolbar itself, only the parent 's onClick will run

Stopping propogation

→ Event handler receive an event object as their only argument. By convention its usually called 'e', which stands for event.
→ You can use the object to read information about the event.
→ That event object also let you stop the propogation. If you want to prevent an event from reaching parent components, you need to call e.stopPropogation().

Eg:
```
function Button({ onClick, children}) {
    return (
        <button onClick={e => {
            e.stopPropogation();
            onClick();
        }}>
            {children}
        </button>
    );
}
export default function Toolbar() {
    return (
        <div className="Toolbar" onClick={() => {
            alert('You clicked the toolbar');
        }}>
```

```
        <Button onClick={() => alert('playing!')}>
          Play movie
        </Button>
        <Button onClick={() => alert('Uploading!')}>
          Upload Image
        </Button>
      </div>
    );
}
```

→ As a result of e.stopPropagation(), clicking on the button now only shows a single alert (from the <button>) rather than the two of them (from the <button> and the parent toolbar).

Managing State

→ State represent data that changes over time
→ State is local and private to the component
→ State changes causes the component to rerender
→ For functional components, use the useState hooks.
→ React function that start with word use are called hooks
→ Hooks should only be used inside components
→ Parent components can pass state down to children via props
→ Lifting state up : share state b/t components by moving it to their closest common ancestor.

State v/s Props

State :

- Local and mutable data within a component
- Initialized within the component
- Can change over time
- Cause re-render when updated
- Managed using usestate in functional components.

Props :

- Passed into component from its parent.
- Read-only (immutable) within the receiving component.
- Allow parent to child component communication
- Changes in props also cause a re-render

~~Uses~~ useState

→ "useState" is a function that lets you add state to functional components in React.

→ State generally ~~refered~~ refers to data or properties that need to be tracked and updated over time.

Eg.

```
import React, {useState} from 'react'
function App() {
    // Declaring a state variable:
    const [count, setCount] = useState(0);
    return (
      <div>
        <p> You clicked {count} times </p>
        <button onClick={() => setCount(count+1)}>
          Click me
        </button>
```

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

Key Points

- **Initial state**: The argument passed to 'useState' is the initial state
- **State Variable and setter function**: 'useState' return an array with two elements the current state and a function to update the state.
- **Re-rendering**: when the state is updated using the setter function, the component re-render to reflect the new state.
- **Multiple state Variable**: You can use multiple 'useState' hooks to manage different pieces of state in the same components.

Example:

```
function UserProfile() {
    const [name, setName] = useState ('Ramlesh');
    const [age, setAge] = useState (22);
    return (
        <div>
            <p> Name: {name} </p>
            <p> Age : {age} </p>
            <button onClick={() => setAge(age+1)}>
                Increase Age
            </button>
        </div>
    );
}
```

PAGE NO.:
DATE: / /

How React Works

⇒ React component :
- The App is the main or root component of a React application
- It's the starting point of your React component tree

⇒ Virtual DOM :
- React creates an in-memory structure called the virtual DOM
- It's a light weight representation where each node stands for a component and its attributes.
- Virtual DOM acts as an intermediate between the actual DOM and the React components, allowing React to make updates to the DOM efficiently. Once the virtual DOM has been updated React updates the actual DOM with the least changes required.
- React compare the current and previous version of the virtual DOM
- It defines the specific nodes that need updating
- Only these nodes are updated in the real browser DOM, making it efficient.

⇒ React and ReactDOM
- The actual updating of the browser's DOM isn't done by react itself
- It's handled by a companion library called react-dom.

⇒ React Elements

- The root div acts as a container for the React App
- The script tag is where the React app starts executing
- If you check main.jsx, the component tree is rendered inside these root element.

⇒ Strict Mode Component :

- It's a special component in React
- Doesn't have a visual representation
- Its purpose is to spot potential issues in your React App.

⇒ Platform Independence :

- React's design allows it to be platform-agnostic
- While react-dom helps build web UIs using React, ReactNative can be used to craft mobile app UIs.

⇒ React, Angular and Vue

- React is a library, while Angular and Vue.js are frameworks.
- React focuses on UI; Angular and Vue.js offer comprehensive tools for full app development.

⇒ Library V/s Framework

- A library offers specific functionality
- A framework provides a set of tools and guidelines.
- In simpler terms : React is a tool; Angular and Vue.js are toolsets.

⇒ About Angular and Vue.js

- Angular, developed by Google, provides a robust framework with a steep learning curve
- Vue.js is known for its simplicity and ease of integration, making its begineer-friendly.

React Forms :

→ React Forms are mostly like normal HTML forms, except we use state in this to handle inputs

Key points :

- State management :- each input state is stored in the component's state.
- Handling changes :- use onChange to detect input changes.
- Submission :- Utilize onSubmit for form submission and prevent default with event.preventDefault()
- Validation :- Implement custom validation or use third party libraries.

Handling forms

→ In React all the data is handled by component and stored in component state. We can change state with event handlers in the onChange attribute.

Eg.
```
import {useState} from 'react';
function form() {
    const [email, setEmail] = useState('');
    return(
      <form>
        <label>
```

```
        Enter your email : <input type = "email"
          value = {email} onChange = {(e) => setEmail
                (e.target.value)} />
        </label>
      </form>
    )
}
```

Submitting form

→ We can submit form with onSubmit attribute for the <form>

Eg.
```
import {useState} from 'react';
function Form() {
    const [email, setEmail] = useState('');
    const handleSubmit = (e) => {
        e.prevent.Default ();
        alert ('Your email is : ${email}')
    }
    return (
        <form onSubmit = {handleSubmit}>
            <label>
                Enter your email : <input type: "email"
                value = {email} onChange = {(e) =>
                setEmail (e.target.value)}/>
            </label>
            <input type = "submit" />
        </form>
    )
}
export default Form;
```

React useRef

→ useRef keeps the value stored b/t renders.
→ useRef allows access to DOM element directly and retain mutable values without re-renders.
→ Used with the ref attribute for direct DOM interaction
→ Can hold previous state or prop value
→ Not limited to DOM references ; can hold any value.
→ Refs can be passed as props also.

```
const refobject = useRef (intitial value)
```

ref: react return an object with the current property

Eg.

```
import {useState, useRef} from "react";
import ReactDOM from "react-dom/client";

function App() {
    const [stateValue, setStateValue] = useState("");
    const refValue = useRef("");

    return (
      <>
        <h1>will cause a re-render :</h1>
        <button onClick={() => {
          setStateValue(Math.random())}}>
        state </button> : {stateValue}
```

```
<h1> Will change but won't cause re-render : </h1>
<button onClick={() => {refValue.current =
    Math.random()}}> ref </button> :
    {refValue.current}
</>
);
}

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

→ In this if you click on state button, state will update and it will re-render, whereas if you click on ref nothing visible will happen i.e ref will change but it wont re-render.

→ To verify this, you can again click on state, it will again update and re-render but this time ref has some value stored so that will be displayed.

Accessing DOM elements with ref

Eg.
```
import {useRef} from "react";
function App()
    const inputElement = useRef();
    const focusInput = () => {
        inputElement.current.focus();
    };
    return (
        <>
            <input type = text ref = {inputElement}/>
            <button onClick = {focusInput}> Focus Input
            </button>
    ); </>
```

→ In this we initialized a useRef. In jsx, we used an attribute called ref to store reference, once we have reference then we can use it anyway we want.

Update state from Previous State

→ Spread Operator : Use to maintain immutability when updating array or objects.

→ functional ~~op~~ updates : Use (existingPosts) ⇒ [postData, ...existingPosts] to avoid stale values during asynchronous updates.

→ Updating state in React using the spread operator is a common practice, especially when dealing with objects or arrays. The spread operators ('...') allows you to create a shallow copy of the existing ~~co~~ state and then update specific properties.

Eg. Updating an Object

```
import React, { useState } from 'react';
function Demo() {
    const [state, setState] = useState({
        name: 'Kamlesh',
        age: 21,
        location: 'Vapi'
    });

const updateLocation = () => {
        setState(prevState => ({ ...prevState,
            location: 'Vadodara'
        }));
};
```

```
return (
    <div>
        <p> Name : {state.name} </p>
        <p> Age  : {state.age} </p>
        <p> Location : {state.location} </p>
        <button onClick={updateLocation}> Move to
            Vadodara </button>
    </div>
);
}
export default Demo;
```

Eg. Updating an Array

```
import React, {useState} from 'react';
function UpdatingArray() {
    const [items, setItems] = useState(['Apple', 'Banana']);
    const addItem = () => {
        setItems(prevItems => [...prevItems, 'Orange']);
    };

    return (
        <div>
            <ul>
                {items.map((item, index) => (
                    <li key={index}> {item} </li>
                ))}
            </ul>
            <button onClick={addItem}> Add Orange
            </button>
        </div>
    );
}
export default UpdatingArray;
```

React Router

→ React router is used for page routing as React App doesn't include it as default.

Add React Router

To install React router, run this in your terminal :

```
npm i -D react-router-dom
```

Creating Multiple Routes

→ To do this first we need to create multiple files and to keep code clean, we will make a folder and make all the pages there, hence we will create a folder named pages in src.

Folder structure :

src/pages/ :

- Home.js
- Blogs.js
- Contact.js

Home.js

```
import {Link} from "react-router-dom";
const Home = () => {
    return (
        <>
            <nav>
                <ul>
                    <li><Link to="/"> Home </Link></li>
                    <li><Link to="/blogs"> Blogs </Link></li>
```

```
            <li>
                <Link to="/contact">Contact</Link>
            </li>
        </ul>
    </nav>
  </>
  )
};
```

Blogs.js

```
const Blogs = () => {
    return <h1>Blogs</h1>
};
export default Blogs
```

Contact.js

```
const Contact = () => {
    return <h1>Contact</h1>
};
export default Contact;
```

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import {BrowserRouter, Routes, Route} from "react-router-dom";
import reportWebVitals from './reportWebVitals';
import Home from "./pages/Home";
import Blogs from "./pages/Blogs";
import Contact from "./pages/Contact";

export default function App() {
    return (
        <BrowserRouter>
            <Routes>
                <Route index element={<Home/>}/>
                <Route path="/blogs" element={<Blogs/>}/>
                <Route path="/contact" element={<Contact/>}/>
```

```
        </Routes>
      </BrowserRouter>
    );
}

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

~~reportWebVitals();~~

React Memo

→ 'React.memo' is a higher order component (HOC) that you can use to optimize functional components by memoizing them.

→ Memoizing helps to prevent unnecessary re-renders by caching the re-render output and skipping rendering if the props have not changed.

How 'React.memo' Works

→ 'React.memo' checks if the props of a component have changed. If they have not, the component is not re-rendered. This can lead to performance improvement, especially in large application with complex component trees.

eg.
```
import React from 'react';
const MyComponent = React.memo(({name}) => {
    console.log('Rendering MyComponent');
    return <div>Hello, {name}!</div>;
});

function App() {
    const [count, setCount] = useState(0);
    return (
        <div>
            <button onClick={() => setCount(count+1)}>
                Increment </button>
            <p> Count : {count} </p>
            <MyComponent name = "kamlesh" />
        </div>
    );
}

export default App;
```

→ Here,

- Component Memoization : 'MyComponent' is wrapped in 'React.memo', which ~~means~~ the memoize the component.
- Rendering condition : 'MyComponent' will only re-render if its 'name' prop changes. In this case, the 'name' prop is "Kamlesh" and does not change, so 'MyComponent' does not re-render when the button is clicked
- Console Log : The console log in 'MyComponent' helps to demonstrate when the component is re-rendered.

Context API

Problem:

→ Suppose there is a button in <Navbar /> component which affects something in <Home /> component. Problems: In this case, state can't be held in <Navbar /> or <Home />, it will be in App.js. To access it, we have to pass it like a prop to the components, and if the component is within 3-4 components, we have to pass it to everyone.

Solution:

→ Context API in React is a powerful feature that allows you to share state and other values across your component tree without having to pass props down manually at every level.

→ It is particularly useful for global state management and for avoiding "prop drilling" (passing props through many level of components).

Basic Concepts

(1.) Context Creation : Use `React.createContext` to create a new context.

(2.) Provider : The context provider component makes the context value available to all nested components.

(3.) Consumer : The context consumer component or `useContext` hook allows components to access the context value.

Eg.

context.js

```
import {createContext} from "react";
export const counterContext = createContext(0)
```

App.jsx

```
import {counterContext} from "./context";
import {useState} from "react";
import comp from "./Comp";
function App() {
    const [count] = useState(25);
    return (
      <>
        <counterContext.Provider value={count}>
            <Comp/>
        </counterContext.Provider>
      </>
    );
}
```

Comp.jsx

```
import {useContext} from 'react'
import {counterContext} from './context';
function Comp(){
    const counter = useContext(counterContext)
    return <div>{counter}</div>
}
```