React Js:
- React Js is Javascript libraray.
- ReactJS, is a popular and powerful JavaScript library used for building dynamic and interactive user interfaces, primarily for single-page applications (SPAs).
- It allows developers to create reusable UI components and manage the state efficiently.
- React is a JavaScript library for building user interfaces (UIs) on the web.
- React is a declarative, component based library that allows developers to build reusable UI components and It follows the Virtual DOM
  (Document Object Model) approach, which optimizes rendering performance by minimizing DOM updates.

==
==========

History of ReactJs:
- Jordan Walke, a software engineer at Facebook, developed the first version of React, initially named FAXJS.
- They made react js as open source in year 2013.

==
============

Characteristics of ReactJS:
1.Component-Based Architecture: React applications are built using small, reusable components.
2.Virtual DOM: React uses a virtual DOM to improve performance by minimizing direct manipulation of the real DOM.
3.Declarative Syntax: React code is declarative, making it easier to read and debug.
4.Uni-directional data flow: Data flows in a single direction, enhancing control and predictability.
5.JSX (JavaScript XML): JSX allows writing HTML-like syntax within JavaScript.
6.Write Once and use Anywhere.

==
=========

Advantages of ReactJS:
- Fast Rendering: The virtual DOM ensures updates are efficient, reducing the load on the actual DOM.
- Reusable Components: Promotes code reusability, reducing development time and effort in maintenance.
- Rich Ecosystem: React provides various tools and libraries, such as React Router and Redux, for seamless development.
- Strong Community Support: React has a large and active developer community, ensuring robust support and resources.

Single page Application Vs Multipage Application:

============= Single Page Application======== || =================== Multipage

Application ===================

1. Having only one webpage	1. Having Multiple webpages
2. Requests are less.	2. Requests are More
3. Rendering time is very less	3. Rendering time is more
4. Example:Instagram,YouTube,Linkdin,FaceBook	4. Example: W3schools,Javatpoint,flipkart

WHAT IS DOM:
- DOM is the tree data structure used by the browser , and it is the representation of structured object in the form of tree structure, any updates in the dom results in re-rendering and repainting of the UI .

WHY React USES Virtual DOM:
- The DOM (Document Object Model) in the browser is like a tree of all the elements on a web page. When you update something, the browser checks and reloads parts of this tree, which can slow down performance. React solves this by using a Virtual DOM.

Virtual DOM:
- The virtual DOM is a lightweight copy of the real DOM that allows React to manage changes more efficiently by minimizing the direct manipulation required on the real DOM
- Virtual DOM is the clone of real DOM.
- When the DOM is loaded than only you can get the data and store into virtual DOM , First time we have to capture the Real DOM and it is done by ReactDOM , Next time onwards it dont use real dom instead of that it use virtual DOM
- No Re Rendering takes place in virtual DOM.

Diffing Algorithm:
- It compares the new virtual DOM with the previous one to identify the differences.
- This comparison is done using a highly optimized diffing algorithm, which minimizes the number of DOM operations required to update the real DOM.

Updating the Real DOM:
- Once the diffing process identifies the necessary changes, React applies them to the real DOM in a batched manner.
- This ensures that the browser only performs the minimum required updates, resulting in improved performance.

Patching:

- Once the diffing algorithm identifies the changes, React generates a set of patches — instructions for updating the real DOM.
- These patches are applied in a batched manner, minimizing the number of DOM manipulations and improving performance

Reconicilation process:
- The process of comparing the previously created virtual DOM copy with the newly created virtual DOM copy using a
  diffing algorithm is known as the reconciliation process.

Steps of reconcilation Process:
- React stores a copy of Browser DOM which is called Virtual DOM.
1.When we make changes or add data, React creates a new Virtual DOM and
  compares it with the previous one.
2.Comparison is done by Diffing Algorithm. The cool fact is all these
  comparisons take place in the memory and nothing is yet changed in the
  Browser.
3.After comparing, React goes ahead and creates a new Virtual DOM having
  the changes. It is to be noted that as many as 200,000 virtual DOM nodes
  can be produced in a second.
4.When the state of a component changes, React compares the VDOM of the
  last and current states and calculates the minimum number of DOM
  operations required to update the actual DOM to match the current VDOM
==

================================= Installation
=====================================

Installing React Using npx Create React App:
  -This is one of the easiest and most common ways to set up a React project.

Steps:
 1.Install Node.js:
  - Download and install Node.js from the official website.
  - Verify the installation by running the following commands in your terminal:
```
node -v
npm -v
```
 2.Create a React Application:
  - Run the following command in your terminal:
```
npx create-react-app my-app
```
  - Replace my-app with your desired project name.

- The npx command ensures you are using the latest version of Create React App without globally installing it.

3.Navigate to Your Project Directory:
- cd my-app

4.Start the Development Server:
- npm start
- Your app will open automatically in the browser at http://localhost:3000/.

Installing React Using Vite:
- Vite is a fast build tool and alternative to Create React App.

Steps:

1.Install Node.js (if not already installed).

2.Create a React Application:
- npm create vite@latest my-app

3.Select Framework and Language:
- When prompted, select the framework as React.

4.Choose the desired variant:
- JavaScript for plain JavaScript.
- TypeScript for TypeScript support.

5.Navigate to Your Project Directory:
- cd my-app

6.Install Dependencies:
- npm install

7.Start the Development Server:
- npm run dev
- Your app will open in the browser at the URL provided by Vite.

==
=======

What is npm?
- Npm means node package manager , this command is only for installing packages , its not for executing packages
- NPM stands for Node Package Manager, and it's a free, open-source tool that helps developers install, manage, and share JavaScript packages.

What is npx?
- Npx means node package extender/Excute , without installing packages you can execute application

React Libraries

1.React
- The "react" is a core library of react.
- It contains functionalities to manage and maintain components, states and handling events.
- import React from "react"

2.ReactDOM
- The "react-dom" library is responsible for rendering the components into the UI/into the DOM tree.
- import ReactDOM from "react-dom/client"

CreateRoot:
- createRoot is a function in React that allows you to create a root for rendering React components within a specified DOM node.
- This method acts as a bridge or connector between index.html and index.js files.
- This method will show the path to React that where all the code has to render.
- Import {createRoot} from react-dom/client library
- Syntax

```
import {createRoot} from "react-dom/client";
createRoot(document.getElementById("root")).render(
    <h1>React 18 onwards use createRoot function ....</h1>
  )

            Or

import ReactDOM from "react-dom/client";
ReactDOM.createRoot(document.getElementById("root")).render(
<h1>React 18 onwards use createRoot function ....</h1>
);
```

createElement:
- The React.createElement method is used to create React elements programmatically.
- It serves as an alternative to using JSX.
- It takes three parameters: the element type (e.g., a string for a DOM tag or a function for a React component), an object containing the
  element's properties (props), and any child elements.
- It returns a plain JavaScript object that represents a virtual DOM node.
- Syntax of createElement:

```
React.createElement(elementtype, property / attribute, ...(children /content));
```

- Example:

```
import {createElement}from "react";
import ReactDOM from "react-dom/client";

let element= createElement("p",{className:"para"},"I am para from react app")
createRoot(document.getElementById("root").render(element)
```

- If you want to render multiple elements created using the React.createElement method and pass them to the createRoot method,
  you typically wrap them inside a single container element.

```
import React from 'react';
import ReactDOM from 'react-dom/client';

// Create multiple elements
const element1 = React.createElement('h1', null, 'Hello, React!');
const element2 = React.createElement('p', null, 'This is a simple example.');


// Wrap elements in a parent container
const container = React.createElement('div', null, element1, element2);

// Render to the DOM
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(container);
```

JSX:
- JSX, or JavaScript XML, is a syntax extension for JavaScript that allows users to write HTML-like markup inside a JavaScript file.
- It is the template language for reactjs
- Jsx means javascript xml or javascript extension
- Jsx allow us to combine html and js in react , basically it is a syntax allowing us to combine html and js in react easily
- It looks similar to html but it is not html.
- Without jsx it is very difficult to create react application
- It is the advance version of javascript if you want to render it you need babel js,
- Babel js is internally converts jsx to browser understandable javascript.

Rules of JSX:
1. Rule1: Close all tags: Explicitly close all tags, including self-closing tags like <img/>.
2. Rule2: In jsx we have to use camelCase convention
3. Rule3: Same level elements if you are having then you should wrap them in a parent element.

```
ReactDOM.createRoot(document.getElementById("root")).render(
                                            <div>
                                               <h1>Hello</h1>
                                               <h1>Hii</h1>
```

```
                                              </div> );
```
4.Rule4: Instead of using class as the attribute , you should use className as the attribute in jsx and Instead of using for attribute of label ,
we should have to use htmlFor here.

```
    Ex:
      <div className="demo" id="demo">
        <label htmlFor="n">Name</label>
        <input type="text" id="n"/>
      </div>
```
5.Rule5: If you want to use multiple lines of JSX than you should have to wrap them in parentheses , otherwise react will throw a warning.

```
    Ex : let list = (
              <ul>
                <li>Raj</li>
                <li>Nikhil</li>
                <li>Suraj</li>
              </ul>
             );
```

React Fragments:

- React Fragment is a feature in React that allows you to return multiple elements from a React component by allowing you to group
  a list of children without adding extra nodes to the DOM.
- It won’t add extra node to the DOM
- It only accept key property , it will not accept any other properties
- Syntax

```
     Ex: //? here in this case this extra node as div has been added to the DOM
          ReactDOM.createRoot(document.getElementById("root")).render(
                        <div> //! adding extra node to the DOM
                           <li>hello</li>
                           <li>world</li>
                           <li>and India</li>
                        </div>
                   );
         // avoiding extra node from adding to the DOM
         import React from "react"
         import {Fragment} from "react"
         ReactDOM.createRoot(document.getElementById("root")).render(
             <React.Fragment key=”1”> //! not adding extra node to the DOM
                  <li>hello</li>
                  <li>world</li>
                  <li>and India</li>
                  </React.Fragment>
```

```
                );
```

- shorthand Fragment component:
    - But here in this shorthand fragment There is a difference
    - You cannot add key property
    - Ex:

```
        <>
            <li>one</li>
            <li>two</li>
            <li>three</li>
        </>
```

JSX Expression ({}):

- A JSX expression is any JavaScript code that is enclosed within curly braces {} inside your JSX elements.
- You can use JSX expressions to dynamically render values, evaluate conditions, and perform calculations within your component's structure.
- Example:

```
        let num1 = 20;
        let num2 = 10;

        ReactDOM.createRoot(document.getElementById("root")).render(
                              <Fragment>
                                 <h1>{num1 + num2}</h1> //! here"{}" is a jsx  expression , because
it is inside jsx
                              </Fragment>
                           );
```

- Example:2

```
        let firstname = "Raj";
        let lastname = "Patil";
        let company = "TYSS";
        ReactDOM.createRoot(document.getElementById("root")).render(
                    <React.Fragment>
                      <h1>my name is {firstname}</h1>
                      <h1>my last name is {lastname}</h1>
                      <p>i work at {company}</p>
                    </React.Fragment>
                   );
```

Components:

- React components are the core building block of the any react application.
- Components are nothing but a block of code, we have to export and import to make

reusability.
 - Webpages will be divided into multiple components (files) and then we will be joining together in the parent component (App.jsx).
 - Components are reusable.

Rules:
 1.Component names should start with a Capital letter only.
 2.Component files have to be saved with .jsx extension (recommended).
 Ex: App.jsx
 3.We can Represent Components in 2 ways
 1. Paired tag: <App> </App>
 2. Self-closing tag: <App/>

- React components can be categorized into the following types:
 1. Function-Based Components (FBC)
 2. Class-Based Components (CBC)

1.Class Based Component
 - React library is mandatory to import. So import React library
 - Use ES6 classes
 - We have to use inheritance (use extends keyword) and we should have to inherit
 - React.Component (React is the base class where we are having a property called as Component)
 - Whenever we are using class based Components , render method is required , without render method your class based components
 will not work , your jsx will not print. This is
 - a lifecycle method. Render methods job is to call sub component and printing jsx. Whatever you write in this render method ,
 for every update this method will rerender
 - Inside render method you can return JSX
 - Class based component is having Lifecycle methods.

```
//! index.js or main.jsx
    import React from react":
    import ReactDOM from 'react-dom/client":
    import App from/App":

    ReactDOM.createRoot (document.getElementById("root")).render (<App />):

// App.ja=sx

   import React, {Component} from "react"
```

```
        class App extends Component {
           render() {
             return <h1> Hii, I am Class Based Component </h1>:
            }
          }
        export default App:
```

- If you console this keyword it will point to the App. you will get state object there. which is defaultly present there in class based components .
  That’s why we call them stateful components

Function-Based Components:
- Function-Based Components (FBC) are JavaScript functions that return JSX elements.
- They are simpler compared to class-based components and are widely used in modern React applications.
- Prior to React Hooks, they were limited to stateless functionalities.
- With the introduction of Hooks (e.g., useState, useEffect), FBCs can now manage state and lifecycle behaviors
- Syntax:

```
       import React from "react";
       const MyComponent = () => {
         return <div>Hello, this is a functional component!</div>;
       };
       export default MyComponent;

                OR
      import React from "react";

      function MyComponent() {
          return <div>Hello, this is a functional component!</div>;
      }
      export default MyComponent;
```

- Both arrow functions and normal functions work for functional components, but arrow functions are more commonly used in modern React.

How to create components ?
- At first we need a main entry point of the react js application which is index.js or main.jsx
- Always one root file is required which is main js file for the react ex → index.js , main .jsx
- Index.html is the main html file for the react . this file is not called as component ,
- We have to create a root component which will be rendered in this root file.

Creating root component:

1. Always in src you can create with extension as js or jsx , ex → App.js pr App.jsx first of all ,after creating root js file
2. Then Import react library only for using some functionalities like hooks and all, we don't need ReactDOM library because ,
   ReactDOM is required only once to specify where we want to render our code
3. Then create a class or function (as we are using functional based components nowadays) and then return the jsx from that function
4. After that export the function
5. And then for using it , you should have to import it . and then compose
6. And then you can pass it to render method

Note: App.jsx root component of the react application , but you can give any name of your choice , not only App.jsx(it should follow naming convention)

Root component
- We have created root component as App.jsx , now we won't be touching index.js.
- We will be rendering our components in root component (App.jsx) .
- We have to use root component to render our components , It will be working as a
- wrapper for all other components

React State:
- State is plain javascript object used by react to represent an information about the component current situation.
- The state is data that changes over time.
- It managed in the component level.
- State is a internal property in react , especially in CBC by default state object will be there
- If you want to use state in function based component . than you should use the hook called as useState() ,
  by using useState() hook you can achieve the state object in FBC.

Hooks:
- Hooks are like inbuilt methods in React.
- Hooks always start with a prefix word "use".
- Hooks, we can use only in Function Based Components.
- Whenever we want to use Hooks,
- We have to import it from the React Library.
- Import Statement is mandatory.
- We have many hooks use State, useEffect, useContext, useCallback and so on.

Rules for hook:
1.Always hooks should be used within a component , not outside a component

2.Hooks always starts with “use”

useState():

- useState is a React Hook that allows functional components to have state.
- It is used to store and update values that change over time within a component.
- When a state changes, React will rerender the component by executing the function that creates the component.

- Syntax:

```
let [state, setState]=useState(initial_value);
```

i. state: It is a variable Name.and in state variable initial_value will be stored

ii. setState:It is a function to change the current value of a state. Its name can be anything.

iii. useState()- It is a Hook to initialize the value.

- The useState() function returns an array that contains two variables:
  i. A state (state).
  ii. A function that changes the state (setState).
- Always use setState() function to change the state.
- Note: By convention, if a state is name, then the function that sets the state is setName.
- useState can hold string , number , boolean , array , object , function. We don't have any restrictions here , we can use any data type in useState

- Example:

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

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

- 1. Initializing State:
  - const [state, setState] = useState(initialValue);
  - `useState` takes an initial state value and returns an array with two elements:
  - The current state (state).
  - A function to update the state (setState).

- 2. Updating State:
  - State updates trigger a re-render of the component.
  - You must use setState to modify the state.

Props:
- In short, we call properties as props.
- Props are used to share the information between the components.
- props is a way of sharing the data from one component to another component. (parent to child) as html attributes.
- Props follow unidirectional flow. i.e from parent component to child component.
- By default, all the props will be stored as objects and passed to the child component.
- Props are immutable, it means once the data is passed from parent component it can't be changed in child component.
- You can send any type of data using props.
- State is for storing data in a component level and Props is for sending Data from parent component to child component
- Syntax:

```
<Child name=”Raj”/>
```

- The data will be passed as an object to the child. We have to go with custom attributes here . we don’t use built-in attributes.
- Child component can only receive the props data but cannot modify/mutate the sender data.
- Props is available in both the components.
- Props is available in function based components also and in class based components also
- Example:

  How to pass props in functional components

```
import Child from "./components/Child";
const Parent = () => {
   return (
           <div>
              {/* Here only for first child we are passing the props , so the props object will
be there in the first child */}
           <Child name="Raj" food="paneer tikka" />
           <Child food="pulav" name="pranav" />
          </div>
       );
};
export default Parent;
```

  How to consume Props in functional based components

```
const Child = props => {
```

```
            console.log(props); // it will return an object containing the both the  props //!{name:
'Raj', food: 'paneer tikka'}
             return (
                   <>
                    <div>
                       My name is {props.name} and i like to eat {props.food}
                    </div>
            </>
           );
           };
             export default Child;
```

Passing different type of props:

- For passing string data we dont need any expression we can send string normally like <Child food="pulav" name="Raj" />
- Except string , if you want to pass the data then you should have to use expression Wrap the data inside an expression and then you have to send
  But we can consume them in the same way

- Ex:

```
     <Child
         name="Raj"
         salary={20000}
         isAvailable={false}
         isNull={null}
         isUndefined={undefined}
     />
```

Consuming data

```
        const Child = props => {
           let { name, salary, isAvailable, isNull, isUndefined } = props;
             return (
                   <>
                    <div>My name is {name}</div>
                    <p>My salary is {salary}</p>
                    <p>is Availble: {isAvailable}</p> {/* here it will not print true because we know
that boolean null or undefined will not be printed on
                                          the UI , boolean we can use for conditional rendering purpose */}
                    <p>{ isAvailable?'yes i am available':'i am not available'}</p>
                    <p>{isNull === null ? "loading" : "some content"}</p>
                    <p> {isUndefined === undefined ? "loading with undefined data" : "we got
content!"}</p>
                   </div>
```

```
                </>
              );
             };
           export default Child;
```

Passing an object a prop:

-
```
const Parent = () => {

      let obj = {
            name: "shashi",
            salary: 10000,
            isAvailable: true,
            isNull: null,
            isUndefined: undefined,
       };
    return (
          <>
             <Child objectProps={obj} />
          </>
      );
 };
  export default Parent

 const Child = props => {
 let { name, salary, isAvailable, isNull, isUndefined} = props.objectProps;
 let obj = new Object(isSymbol);
 console.log(obj);
   return (
          <>
            <div>My name is {name}</div>
            <p>My salary is {salary}</p>
            <p>is Available: {isAvailable}</p>{" "}
            <p>{isAvailable ? "yes i am available" : "i am not available"}</p>
            <p>{isNull === null ? "loading" : "some content"}</p>
            <p> {isUndefined === undefined? "loading with undefined data": "we got
content!"}</p>
            <p>{obj.description}</p>
           </>
         );
    };

    export default Child
```

Default Props:
- In React, default props allow you to specify default values for props that a component might not receive.
- Generally, Props are used to share information between components and it helps to component reusability by passing different props.
- What if a component was made with some prop data and that prop was not present.
- So, we are able to decide what if the prop is not present.
- Now the solution is passing default props.
- If the data has not been sent then the component will consider this data.
- The defaultProps is a react property that allows you to set default values for props objects.
- Example:
```
        const Parent = () => {
             return (
                  <div>
                    <Child2 name="Raj" company="TCS" salary={200000} />
                    <Child2 company="Capgimini" salary={100000} />
                    <Child2 name="Vishal" salary={300000} />
                    <Child2 name="Gunda" company="Infosys" />
                  </div>
                 );
             };

       const Child2 = ({ name, company, salary }) => {
             return (
                   <div>
                     <p>Name is {name}</p>
                     <p>Name is {company}</p>
                     <p>Name is {salary}</p>
                   </div>
                 );
            };
```

- So here is how we can specify default props in a component
    1st Way
      - instead of React.defaultProps you can use a simpler or operator.
```
        const Child2 = (props) => {
            let {name, company, salary}=props
         return (
              <div>
                 <p>Name is {name || "Shubham"}</p>
                 <p>Name is {company || "Test Yantra"}</p>
```

```
                <p>Name is {salary || 800000}</p>
            </div>
        );
    }
```

2nd way

- Passing default values at the time of destructuring.

```
const Child2 = ({ name = "anurag", company = "anything", salary = 50000 }) => {
        return (
            <div>
              <p>Name is {name}</p>
              <p>Name is {company}</p>
              <p>Name is {salary}</p>
            </div>
          );
      };
```

Props drilling :

- Props drilling is a situation in React where you pass data (props) from a parent component to a deeply nested child component through multiple intermediate components, even if they don't need the data themselves
- Example:

```
const App = () => {
    const user = { name: "Raj", age: 30 };
    return <Parent user={user} />;
  };

const Parent = ({ user }) => {
   return <Child user={user} />;
 };

const Child = ({ user }) => {
  return <GrandChild user={user} />;
  };

 const GrandChild = ({ user }) => {
     return <h2>User: {user.name}</h2>;
 };
```

- In this case, user is passed from App → Parent → Child → GrandChild, even though Parent and Child don’t actually use it.

How to Avoid Props Drilling:

1.React Context API :
  - Instead of passing props through multiple layers, you can store the data in a context and access it anywhere.

2.State Management Libraries (Redux, Zustand, Jotai, etc.)
  - If your app grows and needs centralized state management, using Redux or Zustand can help store and retrieve global state without drilling props.

==

Context API:
 - This is Part of react API after 16.2 onwards. Context provides a way to pass the data through the component tree without having to pass props
   down manually at every level.
 - So that if you want to avoid props drilling. You can use context API.
 - Context API is the best way to avoid props drilling.
 - If it is a simple application or moderate application you can use context API .
 - If it is a complex application , you can go with third party libraries like mobx or redux tool.
 - Context API provides us to store global state. Instead of component state. You can store global state.
 - If it is a global state than easily you can access data at any level.
 - Simply you can consume data at whichever component you need

Why do we need Context API ?
  - Already we have a state, so why do we need context API?
  - The state belongs to the component. The state is isolated within the
  - component so the state is not accessible outside the component.
  - If you want to pass that state out of the component ,You can use props
  - Ex→ If we have 10 components then definitely we need props drilling. You need to drill props manually at every level. So definitely this is not
    recommended by react.

How to use Context API:
  - In context API we have two important things
    1. Provider
    2. Consumer
  - to consume the data we need provider. So in provider we need to set global state
  - To set the global state , react library is providing a method called as createContext method

-

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

Step1: create Context API

- First, create a new context using createContext.

```
import { createContext } from "react";
const ThemeContext = createContext(null); // returns an object with properties ad provider and consumer
```

- By using createContext method you can create provider and you can consume provider Data
- createContext will create an state and based on that state you can create provider and you can create n number of providers by using createContext method
- The createContext method is coming from context API

Step 2: create a Provider by using the context you created:

- In provider use value prop, value is the default prop and by using value you
- can set global variable in context API
- Value prop is mandatory
- The provider component wraps around the part of the application that needs access to the context.

```
 const ThemeProvider = ({ children }) => {
 const [theme, setTheme] = useState("light");

  const toggleTheme = () => {
     setTheme((prevTheme) => (prevTheme === "light" ? "dark" : "light"));
   };

 return (
     <ThemeContext.Provider value={{ theme, toggleTheme }}>
       {children}
    </ThemeContext.Provider>
  );
};

export default ThemeProvider;
```

useContext():

- In react 16.8 they introduced a hook called as useContext , this hook is the alternative for Consumer
- This useContext is only available in functional component

Step 3: Use Context in a Component

- Now, access the context in a child component using useContext.

```
import { useContext } from "react";
import ThemeContext from "./ThemeContext";

const ThemeToggleButton = () => {
    const { theme, toggleTheme } = useContext(ThemeContext);

  return (
      <button onClick={toggleTheme} style={{ background: theme === "light" ? "#fff" :
"#333", color: theme === "light" ? "#000" : "#fff" }}>
        Toggle Theme
      </button>
   );
};
export default ThemeToggleButton;
```

Step 4: Wrap Application with Provider

- Ensure the provider wraps the components that need access to the context.

```
import React from "react";
import ThemeProvider from "./ThemeProvider";
import ThemeToggleButton from "./ThemeToggleButton";
const App = () => (
      <ThemeProvider>
          <ThemeToggleButton />
      </ThemeProvider>
);
```

React Conditional Rendering:

- Conditional rendering in react works the same way as conditions in javascript.
- We use conditional rendering to print content on UI based on condition.
- In react you can conditionally render jsx using js syntax like if else condition or switch case statements, logical and operator ot ternary operator

1. First way to use conditional rendering is by using if else:

```
const Conditional1 = () => {
      let [displayText, setDisplayText] = useState(true);
      if (displayText) {
          return (
              <>
                  <h1>Welcome to Testyantra software solution pvt ltd</h1>
```

```
                              <p> Lorem ipsum dolor, sit amet consectetur adipisicing elit. Ducimus
laudantium distinctio natus nulla est. Dicta modi,</p>
                         </>
                       );
             } else {
              return (
                      <>
                          <h1>no data found</h1>
                     </>
                   );
                 }
             };
```

2. Using switch case statements

```
             import React, { useState } from "react";
             const ProductBlock = () => {
                  const [mode, setMode] = useState("a");
                  const [color, setColor] = useState("green");

                   if (mode === "a" && color === "red"){
                           return <h1 style={{ color }}>Mode is A</h1>;
                     }
                   if (mode === "b" && color === "green") return <h1 style={{ color }}>Mode is B</h1>;
                   if (mode === "c" && color === "yellow") return <h1 style={{ color }}>Mode is
C</h1>;

                };
            };
           export default ProductBlock;
```

3.conditional rendering using the ternary operator in React:

```
          import React, { useState } from "react";
         const ToggleMessage = () => {
             const [isVisible, setIsVisible] = useState(true);
              return (
                    <div>
                         <h1>{isVisible ? "Hello, Welcome!" : "Goodbye, See you soon!"}</h1>
                          <button onClick={() => setIsVisible(!isVisible)}>
                         {isVisible ? "Hide Message" : "Show Message"}
                         </button>
                  </div>
  );
 };
```

export default ToggleMessage;

4.Short Circuit AND Operator &&
- When you want to render either something or nothing you can use and operator, unlike In ternary either if block executes or else block executes.
- Example:

```
    import React, { useState } from "react";
    const WelcomeMessage = () => {
      const [isLoggedIn, setIsLoggedIn] = useState(false);
    return (
       <div>
        {isLoggedIn && <h1>Welcome Back, User!</h1>}
        <button onClick={() => setIsLoggedIn(!isLoggedIn)}>
         {isLoggedIn ? "Logout" : "Login"}
       </button>
     </div>
     );
  };
  export default WelcomeMessage;
```

- The && operator ensures that the <h1> element only renders if isLoggedIn is true.
- If isLoggedIn is false, nothing is displayed.
- Clicking the button toggles the login state.

CSS in React JS:
- In React JS, we can achieve CSS in 3 woys.
- They are
  1.Inline CSS
  2.Internal CSS
  3.Global CSS
  4.Module CSS

1. Inline CSS:
  - In inline CSS, we have to use an attribute style...
  - In style attribute, It accepts an object as a value.
  - CSS properties should be in lower camel Cose.
  - CSS properties should be written inside an expression in the form of "object".

2. Internal CSS:
  - Internal CSS in React involves defining CSS rules within your component file, but outside the JSX returned by the render method.
  - It leverages JavaScript's ability to create style objects.
  - You then apply these styles using the style prop on your JSX elements.

- How it works:
    1.Define Style Objects: Create JavaScript objects where the keys are CSS properties (in camelCase) and the values are the corresponding CSS values.
    2.Apply Styles: Use the style prop on JSX elements. Assign the style object you created to this prop.
- Example:

```
          import React from 'react';

           function MyComponent() {
                     const myStyle = {
                     color: 'blue',
                     fontSize: '20px',
                     backgroundColor: 'lightgray',
                     padding: '10px'
                     };

                     const anotherStyle = {
                       color: 'red',
                      fontWeight: 'bold'
                     }
            return (
              <div>
                  <h1 style={myStyle}>This is a heading with internal styles.</h1>
                  <p style={anotherStyle}>This is another paragraph.</p>
                  <p style={{ ...myStyle, ...anotherStyle}}>This paragraph has combined styles.</p>
{/* Combining styles using spread syntax */}
              </div>
            );
           }
         export default MyComponent;
```

3. Global CSS:
  - This CSS code will opply for the entire project.
  - First, we have to create a CSS file with .css extension.
  - Next, we have to import the CSS globally in the index.js or mains.jsx file

4. Module CSS:
  - If we want to apply CSS for a particular set of components we have to use Module CSS.
  - First, we have to create a new file with filename.module.css Extension.
  - Next, we have to import the CSS file into that particular component.
  - If we want to provide any id or class name, we have to provide it with respect to importing variable name

5. Tailwind CSS:
  - Installing Tailwind CSS as a Vite.
  - 1.Install Tailwind CSS
    Install tailwindcss and @tailwindcss/vite via npm.
  - 2.Configure the Vite plugin
    - Add the @tailwindcss/vite plugin to your Vite configuration.
    - In vite.config.js

```
import { defineConfig } from 'vite'
import tailwindcss from '@tailwindcss/vite'
export default defineConfig({
   plugins: [
         tailwindcss(),
          ],
   })
```

  3.Import Tailwind CSS
    - Add an @import to your CSS file that imports Tailwind CSS.
    - In index.css

```
@import "tailwindcss";
```

React Refs:
 - Ref means reference.

what is this reference? .
 - See React is a frontend library so definitely you should interact with dom
 - Refs provide a way to access DOM nodes or React elements created in the render method.
 - React refs is providing us to connect native DOM elements and react elements Without having virtual DOM
 - But it is recommended not to overuse react references because it is effectingour application , because this is imperative,

  without virtual DOM means it
 - will not provide any optimization or performance thing.
 - This is like your normal DOM.
 - There are a few good use cases for refs:
   - Managing focus, text selection, or media playback.
   - Triggering imperative animations.
   - Integrating with third-party DOM libraries.

 - Because it is directly interacting with DOM elements , imperatively it is connecting to the DOM. it is not declarative.
 - But you can connect native elements through refs

useRef():
 - In react 16.8 onwards they introduced one more hook called useRef.

- Either you can use createRef() or you can use useRef() .
- useRef() is only available in FBC.
- In React, the useRef hook allows you to access a DOM element directly like document.querySelector() in plain JavaScript.
- Additionally, the useRef hook lets you modify a state without causing a re-render.
- Here are the steps for using the useRef hook.

Step 1. Import useRef from React:

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

Step 2. Creating the ref object:

- Call the useRef() function with an initial value to create a ref object:

```
const myRef = useRef(initialValue);
```

- The return type of the useRef() function is a mutable object MyRef.

Step 3. Attach the ref object to a DOM element.

- Attach the ref to a DOM element using the ref attribute if you want to access the DOM element directly.

```
<input ref={myRef} type="text" />
```

Step 4. Use the ref object.

- Access the current value of the ref via the .current property inside the event handlers.

```
const handleClick = () => {
    console.log(myRef.current.value);
  };
```

Step 5. Update the ref

- Update the .current property of the ref object to change its value without causing a re-render:

```
myRef.current = newValue;
```

useRef vs. useState hook:

-The following table highlights the differences between useRef and useState hooks:

Feature	useRef	useState
Purpose	To persist a mutable value without causing re-renders	To manage state in a functional component that causes re-renders when updated.
Initial Value	useRef(initialValue)	useState(initialValue)
Return Value	An object with a .current property.	An array with the current state value and a function to update it.

Re-renders	Does not cause re-renders when .current is updated.	Causes re-renders when the state is updated.
Use Case	Accessing DOM elements directly, storing mutable values, and holding previous values.	Managing component state that affects rendering.
Value Persistence	Persists between renders.	Persists between renders.
Update Method	Directly update .current property.	Use the setter function provided by useState

React Forms:
- We have two ways to handle forms in react js
  a.Controlled way
  b.Uncontrolled way

1.Uncontrolled Component:
- For uncontrolled components react doesn’t use state Thus uncontrolled components do not depends on any state of input elements or any event handler.
- This type of component doesn’t care about real time input changes.
- Uncontrolled components are form inputs that manage their own state internally, rather than being controlled by React state.
- You can access the current value of the input using a ref after the form is submitted or whenever needed
- Ref attribute is mandatory.
- Here we don’t need any handler except submit
- Uncontrolled means you cannot control by react.
- Form data is handled by the Document Object Model (DOM) rather than by React. The DOM maintains the state of form data and updates it based on user input.

Internal State:
- Each form element in the DOM inherently maintains its own state. For example:
- An <input> element keeps track of its value attribute internally.
- A <textarea> element holds its text content.
- A <select> element maintains the currently selected <option>.
- When a user interacts with these elements (typing in an input box, selecting an option, etc.), the browser updates this internal state automatically.
- You can access the current value of a DOM element via JavaScript using the element's

properties (.value, .checked, etc.). In React, you typically use refs to
access these values

How to handle uncontrolled components:

- By using ref you can handle uncontrolled component , means this is not react data flow , this id DOM data flow Means uncontrolled component managed by components own internal state , that is DOM internal state rather than react state .
- Uncontrolled components doesn't have any react state. Data flow within the component
- By using ref you can create uncontrolled component .
- These are purely imperative by using ref . Uncontrolled components are normal low level components
- Example:

```
    import { useRef } from "react";
    const Uncontrolled = () => {
        let emailRef = useRef();
        let passwordRef = useRef();
        let handleSubmit = e => {
               e.preventDefault();
            let email = emailRef.current.value;
            let password = passwordRef.current.value;
            console.log({ email, password });
          };
       return (
            <div>
              <section id="form">
                 <article>
                        <h2>login</h2>
                        <form>
                          <div className="form-grouop">
                              <label htmlFor="email">Email</label>
                              <input ref={emailRef} type="email" placeholder="email" id="email"/>
                          </div>
                         <div className="form-grouop">
                              <label htmlFor="password">Password</label>
                              <input ref={passwordRef} type="password" placeholder="password"
id="password"/>
                         </div>
                         <div className="form-grouop">
                              <button onClick={handleSubmit}>Login</button>
                         </div>
                       </form>
                 </article>
               </section>
            </div>
          );
        };
      export default Uncontrolled;
```

Controlled Components:
  - In this approach, form data is handled by React through the use of hooks such as the useState hook.
  - In React, a controlled component is a component where form elements derive their value from a React state.
  - When a component is controlled, the value of form elements is stored in a state, and any changes made to the value are immediately
    reflected in the state.
  - To create a controlled component, you need to use the value prop to set the value of form elements and the onChange event to handle
    changes made to the value.
  - The value prop sets the initial value of a form element, while the onChange event is triggered whenever the value of a form element
   changes. Inside the onChange event, you need to update the state with the new value using a state update function.

Creating controlled component:
  - Initialize state object for each input you need
  - Add value attribute to the input or form elements inside which we pass respective state for updations
  - State mutation(updation) add onChange event to the element

onChange:
  - this is a keyboard event, anything changes in your input this event will be triggering.
  - The change event occurs when the value of an element has been changed (only works on <input>, <textarea> and <select> elements).
  - The change() method triggers the change event, or attaches a function to run when a change event occurs.
  - Note: For select menus, the change event occurs when an option is selected.4

How to get the data:
  - Inside the event object we have a property called target to get our targeted element.
      Syntax - e.target
  - To get the value we have to use value property on targeted element (present in input elements only)
      Syntax - e.target.value
  - To get the name of our input element we have to use name property present on targeted element (present in input element only)

Syntax - e.target.name
- Inside the onChange event update the state of that input element by getting the value what user has entered.
  To get the value use → e.target.value
- Don't add a click event on your button, instead add a submit event on your form and then you can get your data using the state's which you
  have created for each respective element.
- Example:

```
    import React, { useState } from "react";
     // initialize state object
     // add value attribute to the input of form elements and assign with initial state
     // state mutation add onChange event to the elements
    const Controlled = () => {
         let [email, setEmail] = useState("");
         let [password, setPassword] = useState("");
         let handleSubmit = e => {
          e.preventDefault();
          console.log({ email, password });
          setEmail("");
          setPassword("");
           };
    return (
          <div>
             <h2>Login</h2>
             <form onSubmit={handleSubmit}>
               <input  type="email" placeholder="enter email" value={email} onChange={e => {
setEmail(e.target.value)}} />
               <input  type="password" placeholder="enter password" value={password}
onChange={e => setPassword(e.target.value)}/>
               <button>Login</button>
             </form>
          </div>
        );
    };
    export default Controlled;
```

In FBC using single function to handle change event:
- React controlled form that includes text, email, checkbox, radio, select box, and range inputs, all managed using a single handler with useState
- Example:

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

function ControlledForm() {
```

```
const [formData, setFormData] = useState({
  name: "",
  email: "",
  subscribe: false,
  gender: "",
  country: "india",
  age: 25,
});

const handleChange = (event) => {
  const { name, type, value, checked } = event.target;
  setFormData(() => ({
    ...prev,
    [name]: type === "checkbox" ? checked : value,
  }));
};

const handleSubmit = (event) => {
  event.preventDefault();
  console.log("Form Data Submitted:", formData);
  alert(JSON.stringify(formData, null, 2));
};

return (
  <form onSubmit={handleSubmit}>
    {/* Text Input */}
    <label>
      Name:
      <input type="text" name="name" value={formData.name} onChange={handleChange} />
    </label>
    <br />

    {/* Email Input */}
    <label>
      Email:
      <input type="email" name="email" value={formData.email} onChange={handleChange} />
    </label>
    <br />

    {/* Checkbox */}
    <label>
      Subscribe:
      <input type="checkbox" name="subscribe" checked={formData.subscribe}
onChange={handleChange} />
      </label>
      <br />

      {/* Radio Buttons */}
      <label>Gender:</label>
      <input type="radio" name="gender" value="male" checked={formData.gender === "male"}
onChange={handleChange} /> Male
      <input type="radio" name="gender" value="female" checked={formData.gender ===
"female"} onChange={handleChange} /> Female
      <br />

      {/* Select Dropdown */}
      <label>
        Country:
        <select name="country" value={formData.country} onChange={handleChange}>
          <option value="india">India</option>
          <option value="usa">USA</option>
          <option value="canada">Canada</option>
        </select>
      </label>
      <br />

      {/* Range Input */}
      <label>
        Age: {formData.age}
        <input type="range" name="age" min="18" max="60" value={formData.age}
onChange={handleChange} />
      </label>
      <br />

      <button type="submit">Submit</button>
    </form>
  );
}

export default ControlledForm;
```

React Conditional Rendering:

- Conditional rendering in react works the same way as conditions in javascript.
- We use conditional rendering to print content on UI based on condition.
- In react you can conditionally render jsx using js syntax like if else condition or switch case

statements, logical and operator ot ternary operator

1. First way to use conditional rendering is by using if else:
```
    const Conditional1 = () => {
        let [displayText, setDisplayText] = useState(true);
        if (displayText) {
            return (
                <>
                    <h1>Welcome to Testyantra software solution pvt ltd</h1>
                    <p> Lorem ipsum dolor, sit amet consectetur adipisicing elit. Ducimus
laudantium distinctio natus nulla est. Dicta modi,</p>
                </>
              );
       } else {
        return (
              <>
                  <h1>no data found</h1>
              </>
            );
          }
       };
```

2. Using switch case statements
```
        import React, { useState } from "react";
        const ProductBlock = () => {
            const [mode, setMode] = useState("a");
            const [color, setColor] = useState("green");

            if (mode === "a" && color === "red"){
                  return <h1 style={{ color }}>Mode is A</h1>;
              }
            if (mode === "b" && color === "green") return <h1 style={{ color }}>Mode is B</h1>;
            if (mode === "c" && color === "yellow") return <h1 style={{ color }}>Mode is
C</h1>;

          };
       };
      export default ProductBlock;
```

3.conditional rendering using the ternary operator in React:
```
      import React, { useState } from "react";
     const ToggleMessage = () => {
        const [isVisible, setIsVisible] = useState(true);
```

```
            return (
                  <div>
                       <h1>{isVisible ? "Hello, Welcome!" : "Goodbye, See you soon!"}</h1>
                        <button onClick={() => setIsVisible(!isVisible)}>
                       {isVisible ? "Hide Message" : "Show Message"}
                       </button>
                </div>
 );
};
export default ToggleMessage;
```

4.Short Circuit AND Operator &&
 - When you want to render either something or nothing you can use and operator, unlike In ternary either if block executes or else block executes.
 - Example:

```
          import React, { useState } from "react";
          const WelcomeMessage = () => {
            const [isLoggedIn, setIsLoggedIn] = useState(false);
          return (
             <div>
               {isLoggedIn && <h1>Welcome Back, User!</h1>}
              <button onClick={() => setIsLoggedIn(!isLoggedIn)}>
                {isLoggedIn ? "Logout" : "Login"}
             </button>
           </div>
           );
        };
        export default WelcomeMessage;
```

 - The && operator ensures that the <h1> element only renders if isLoggedIn is true.
 - If isLoggedIn is false, nothing is displayed.
 - Clicking the button toggles the login state.

CSS in React JS:
 - In React JS, we can achieve CSS in 3 woys.
 - They are
   1.Inline CSS
   2.Internal CSS
   3.Global CSS
   4.Module CSS

1. Inline CSS:
   - In inline CSS, we have to use an attribute style...

- In style attribute, It accepts an object as a value.
- CSS properties should be in lower camel Cose.
- CSS properties should be written inside an expression in the form of "object".

2. Internal CSS:
- Internal CSS in React involves defining CSS rules within your component file, but outside the JSX returned by the render method.
- It leverages JavaScript's ability to create style objects.
- You then apply these styles using the style prop on your JSX elements.
- How it works:
    1.Define Style Objects: Create JavaScript objects where the keys are CSS properties (in camelCase) and the values are the corresponding CSS values.
    2.Apply Styles: Use the style prop on JSX elements. Assign the style object you created to this prop.
- Example:

```
        import React from 'react';

         function MyComponent() {
                 const myStyle = {
                 color: 'blue',
                 fontSize: '20px',
                 backgroundColor: 'lightgray',
                 padding: '10px'
                 };

                 const anotherStyle = {
                   color: 'red',
                  fontWeight: 'bold'
                 }
          return (
            <div>
                <h1 style={myStyle}>This is a heading with internal styles.</h1>
                <p style={anotherStyle}>This is another paragraph.</p>
                <p style={{ ...myStyle, ...anotherStyle}}>This paragraph has combined styles.</p>
{/* Combining styles using spread syntax */}
            </div>
          );
         }
       export default MyComponent;
```

3. Global CSS:
- This CSS code will opply for the entire project.
- First, we have to create a CSS file with .css extension.

- Next, we have to import the CSS globally in the index.js or mains.jsx file

4. Module CSS:
   - If we want to apply CSS for a particular set of components we have to use Module CSS.
   - First, we have to create a new file with filename.module.css Extension.
   - Next, we have to import the CSS file into that particular component.
   - If we want to provide any id or class name, we have to provide it with respect to importing variable name

5. Tailwind CSS:
   - Installing Tailwind CSS as a Vite.
   - 1.Install Tailwind CSS
     Install tailwindcss and @tailwindcss/vite via npm.
   - 2.Configure the Vite plugin
     - Add the @tailwindcss/vite plugin to your Vite configuration.
     - In vite.config.js

```
import { defineConfig } from 'vite'
import tailwindcss from '@tailwindcss/vite'
export default defineConfig({
   plugins: [
        tailwindcss(),
         ],
   })
```

   3.Import Tailwind CSS
     - Add an @import to your CSS file that imports Tailwind CSS.
     - In index.css

```
@import "tailwindcss";
```

React Refs:
- Ref means reference.

what is this reference? .
- See React is a frontend library so definitely you should interact with dom
- Refs provide a way to access DOM nodes or React elements created in the render method.
- React refs is providing us to connect native DOM elements and react elements Without having virtual DOM
- But it is recommended not to overuse react references because it is effectingour application , because this is imperative,

  without virtual DOM means it
- will not provide any optimization or performance thing.
- This is like your normal DOM.
- There are a few good use cases for refs:
  - Managing focus, text selection, or media playback.

- Triggering imperative animations.
- Integrating with third-party DOM libraries.

- Because it is directly interacting with DOM elements , imperatively it is connecting to the DOM. it is not declarative.
- But you can connect native elements through refs

useRef():
- In react 16.8 onwards they introduced one more hook called useRef.
- Either you can use createRef() or you can use useRef() .
- useRef() is only available in FBC.
- In React, the useRef hook allows you to access a DOM element directly like document.querySelector() in plain JavaScript.
- Additionally, the useRef hook lets you modify a state without causing a re-render.
- Here are the steps for using the useRef hook.

Step 1. Import useRef from React:

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

Step 2. Creating the ref object:

- Call the useRef() function with an initial value to create a ref object:

```
const myRef = useRef(initialValue);
```

- The return type of the useRef() function is a mutable object MyRef.

Step 3. Attach the ref object to a DOM element.

- Attach the ref to a DOM element using the ref attribute if you want to access the DOM element directly.

```
<input ref={myRef} type="text" />
```

Step 4. Use the ref object.

- Access the current value of the ref via the .current property inside the event handlers.

```
const handleClick = () => {
     console.log(myRef.current.value);
  };
```

Step 5. Update the ref

- Update the .current property of the ref object to change its value without causing a re-render:

```
myRef.current = newValue;
```

useRef vs. useState hook:
-The following table highlights the differences between useRef and useState hooks:

Feature	useRef	useState
Purpose	To persist a mutable value without causing re-renders	To manage state in a functional component that causes re-renders when updated.
Initial Value	useRef(initialValue)	useState(initialValue)
Return Value	An object with a .current property.	An array with the current state value and a function to update it.
Re-renders	Does not cause re-renders when .current is updated.	Causes re-renders when the state is updated.
Use Case	Accessing DOM elements directly, storing mutable values, and holding previous values.	Managing component state that affects rendering.
Value Persistence	Persists between renders.	Persists between renders.
Update Method	Directly update .current property.	Use the setter function provided by useState

React Forms:
- We have two ways to handle forms in react js
    a.Controlled way
    b.Uncontrolled way

1.Uncontrolled Component:
- For uncontrolled components react doesn't use state Thus uncontrolled components do not depends on any state of input
  elements or any event handler.
- This type of component doesn't care about real time input changes.
- Uncontrolled components are form inputs that manage their own state internally, rather than being controlled by React state.
- You can access the current value of the input using a ref after the form is submitted or whenever needed
- Ref attribute is mandatory.
- Here we don't need any handler except submit
- Uncontrolled means you cannot control by react.
- Form data is handled by the Document Object Model (DOM) rather than by React. The DOM maintains the state of form data and updates it based on

user input.
Internal State:
- Each form element in the DOM inherently maintains its own state. For example:
- An <input> element keeps track of its value attribute internally.
- A <textarea> element holds its text content.
- A <select> element maintains the currently selected <option>.
- When a user interacts with these elements (typing in an input box, selecting an option, etc.), the browser updates this internal state automatically.
- You can access the current value of a DOM element via JavaScript using the element's properties (.value, .checked, etc.). In React, you typically use refs to access these values

How to handle uncontrolled components:
- By using ref you can handle uncontrolled component , means this is not react data flow , this id DOM data flow Means uncontrolled component managed by components own internal state , that is DOM internal state rather than react state .
- Uncontrolled components doesn’t have any react state. Data flow within the component
- By using ref you can create uncontrolled component .
- These are purely imperative by using ref . Uncontrolled components are normal low level components
- Example:

```
    import { useRef } from "react";
    const Uncontrolled = () => {
          let emailRef = useRef();
          let passwordRef = useRef();
          let handleSubmit = e => {
                   e.preventDefault();
                let email = emailRef.current.value;
                let password = passwordRef.current.value;
                console.log({ email, password });
            };
          return (
                 <div>
                   <section id="form">
                      <article>
                              <h2>login</h2>
                              <form>
                                <div className="form-grouop">
                                     <label htmlFor="email">Email</label>
                                     <input ref={emailRef} type="email" placeholder="email" id="email"/>
                                </div>
                               <div className="form-grouop">
                                     <label htmlFor="password">Password</label>
                              <input ref={passwordRef} type="password" placeholder="password"
id="password"/>
                          </div>
                          <div className="form-grouop">
                              <button onClick={handleSubmit}>Login</button>
                          </div>
                        </form>
                  </article>
                </section>
              </div>
            );
          };
       export default Uncontrolled;
```

Controlled Components:
  - In this approach, form data is handled by React through the use of hooks such as the useState hook.
  - In React, a controlled component is a component where form elements derive their value from a React state.
  - When a component is controlled, the value of form elements is stored in a state, and any changes made to the value are immediately
    reflected in the state.
  - To create a controlled component, you need to use the value prop to set the value of form elements and the onChange event to handle
    changes made to the value.
  - The value prop sets the initial value of a form element, while the onChange event is triggered whenever the value of a form element
   changes. Inside the onChange event, you need to update the state with the new value using a state update function.

Creating controlled component:
  - Initialize state object for each input you need
  - Add value attribute to the input or form elements inside which we pass respective state for updations
  - State mutation(updation) add onChange event to the element

onChange:
  - this is a keyboard event, anything changes in your input this event will be triggering.
  - The change event occurs when the value of an element has been changed (only works on <input>, <textarea> and <select> elements).
  - The change() method triggers the change event, or attaches a function to run when a change event occurs.
  - Note: For select menus, the change event occurs when an option is selected.4

How to get the data:

- Inside the event object we have a property called target to get our targeted element.
  Syntax - e.target
- To get the value we have to use value property on targeted element (present in input elements only)
  Syntax - e.target.value
- To get the name of our input element we have to use name property present on targeted element (present in input element only)
  Syntax - e.target.name
- Inside the onChange event update the state of that input element by getting the value what user has entered.
  To get the value use → e.target.value
- Don't add a click event on your button, instead add a submit event on your form and then you can get your data using the state's which you
  have created for each respective element.
- Example:

```
          import React, { useState } from "react";
           // initialize state object
           // add value attribute to the input of form elements and assign with initial state
           // state mutation add onChange event to the elements
         const Controlled = () => {
               let [email, setEmail] = useState("");
               let [password, setPassword] = useState("");
               let handleSubmit = e => {
                e.preventDefault();
                console.log({ email, password });
                setEmail("");
                setPassword("");
                 };
          return (
                  <div>
                      <h2>Login</h2>
                      <form onSubmit={handleSubmit}>
                        <input  type="email" placeholder="enter email" value={email} onChange={e => {
setEmail(e.target.value)}} />
                        <input  type="password" placeholder="enter password" value={password}
onChange={e => setPassword(e.target.value)}/>
                        <button>Login</button>
                      </form>
                  </div>
              );
          };
```

```
export default Controlled;
```

In FBC using single function to handle change event:
- React controlled form that includes text, email, checkbox, radio, select box, and range inputs, all managed using a single handler with useState
- Example:

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

function ControlledForm() {
  const [formData, setFormData] = useState({
    name: "",
    email: "",
    subscribe: false,
    gender: "",
    country: "india",
    age: 25,
  });

  const handleChange = (event) => {
    const { name, type, value, checked } = event.target;
    setFormData(() => ({
      ...prev,
      [name]: type === "checkbox" ? checked : value,
    }));
  };

  const handleSubmit = (event) => {
    event.preventDefault();
    console.log("Form Data Submitted:", formData);
    alert(JSON.stringify(formData, null, 2));
  };

  return (
    <form onSubmit={handleSubmit}>
      {/* Text Input */}
      <label>
        Name:
        <input type="text" name="name" value={formData.name} onChange={handleChange} />
      </label>
      <br />

      {/* Email Input */}
      <label>
```

```
        Email:
        <input type="email" name="email" value={formData.email} onChange={handleChange} />
      </label>
      <br />

      {/* Checkbox */}
      <label>
        Subscribe:
        <input type="checkbox" name="subscribe" checked={formData.subscribe}
onChange={handleChange} />
      </label>
      <br />

      {/* Radio Buttons */}
      <label>Gender:</label>
      <input type="radio" name="gender" value="male" checked={formData.gender === "male"}
onChange={handleChange} /> Male
      <input type="radio" name="gender" value="female" checked={formData.gender ===
"female"} onChange={handleChange} /> Female
      <br />

      {/* Select Dropdown */}
      <label>
        Country:
        <select name="country" value={formData.country} onChange={handleChange}>
          <option value="india">India</option>
          <option value="usa">USA</option>
          <option value="canada">Canada</option>
        </select>
      </label>
      <br />

      {/* Range Input */}
      <label>
        Age: {formData.age}
        <input type="range" name="age" min="18" max="60" value={formData.age}
onChange={handleChange} />
      </label>
      <br />

      <button type="submit">Submit</button>
    </form>
  );
```

```
}

export default ControlledForm;
```