

# CHAPTER 7

React JS

## ABSTRACT

This chapter introduces essential React Hooks and tools for building interactive, stateful applications. It covers `useState` for managing local state, `useReducer` for complex logic, and `useContext` for sharing data globally. The `useEffect` hook is explained for handling side effects like API calls. The chapter also explores handling forms using controlled components and state, including basic validation. Finally, it introduces `Axios` as a simple, promise-based HTTP client for fetching and sending data.

## Hooks in react js

- ✓ Hooks were added to React in version 16.8.
- ✓ Hooks allow function components to have access to state and other React features. Because of this, class components are generally no longer needed.

### There are 3 rules for hooks:

- ✓ Hooks can only be called inside React function components (Only call hooks from React functions" — don't call hooks from plain JavaScript functions so that stateful logic stays with the component.)
- ✓ Hooks can only be called at the top level of a component.
- ✓ Hooks cannot be conditional (don't call hooks from inside loops, conditions, or nested statements so that the hooks are called in the same order each render.)

### Why Use Hooks?

- ✓ Makes code **cleaner and reusable**
- ✓ Eliminates the need for **class components**
- ✓ Encourages **functional programming** in React

### You must import Hooks from react.

Suppose, we are using the **useState** Hook to keep track of the application state. Then we have to import it as below.

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

Function/Hook	Purpose	Syntax
<b>useState</b>	Add and manage <b>local state</b> in function components	<code>const [state, setState] = useState(initialValue)</code>
<b>useReducer</b>	Manage <b>complex state logic</b> with a reducer function	<code>const [state, dispatch] = useReducer(reducer, initialState)</code>
<b>createContext</b>	Create a <b>global context</b> to share data across components	<code>const MyContext = createContext(defaultValue)</code>
<b>useContext</b>	Access the nearest <b>Context Provider's value</b>	<code>const value = useContext(MyContext)</code>
<b>useEffect</b>	Run <b>side effects</b> (fetching data, timers, etc.)	<code>useEffect(() =&gt; { /* effect */ }, [deps])</code>

## 1.useState

The React useState Hook allows us to track state in a function component.

State generally refers to data or properties that need to be tracking in an application.

### Initialize useState

We initialize our state by calling useState in our function component.

useState accepts an initial state and returns two values:

- o The current state.
- o A function that updates the state.

### Syntax:

```
Const [current state, function to update state] =useState (initial state)
```

### To import useState

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

## Example:

Write a program to build React app having a button which increase count by 1 while clicking it.

### US1.js

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

```
function US1() {
```

```
  // Declare a new state variable, which we'll call "count"
```

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

```
  function handleCount() { setCount(count+1) }
```

```
  return (
```

```
    <div>
```

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

```
      <button onClick={handleCount}>
```

```
        Click me
```

```
      </button>
```

```
    </div>
```

```
);
}
export default US1
```

### Or inline function calling

```
import { useState } from "react";
function US1 () {
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click me</button>
    </div>
  );
}
export default US1
```

## Example:

Create a program to build React app having buttons to increment and decrement the number by clicking that respective button. Also, increment of the number should be performed only if number is less than 10 and decrement of the number should be performed if number is greater than 0.

### US.js

```
import { useState } from "react";
function US () {
  const [num,setnum]=useState(0)
  function increment(){
    if(num<10){
      setnum(num+1);
    }else{
      return false;
    }
  }
  function decrement(){
    if(num > 0){
```

```

    setnum(num-1);
  }else{
    return false;
  }
}
return (
  <div>
    <button onClick={increment}>Increment</button>
    <button onClick={decrement}>Decrement</button>
    <h1> {num} </h1>
  </div>
)
}
export default US

```

## Example:

Write a program to build React app to perform the tasks as asked below.

- Add three buttons “Change Text”, “Change Color”, “Hide/Show”.
- Add heading “LJ University” in red color(initial) and also add “React Js Hooks” text in h2 tag.
- By clicking on “Change text” button text should be changed to “Welcome students” and vice versa.
- By clicking on “Change Color” button change color of text to “blue” and vice versa.  
**This color change should be performed while double clicking on the button.**
- Initially button text should be “Hide”. While clicking on it the button text should be changed to “Show” and text “React Js Hooks” will not be shown.

### US1.js

```

import {useState} from "react";

function US1(){
  //useState to Change text
  const [name,setName] = useState("LJ University");

```

```
//useState to Change Color
const [textColor,setcolor] = useState("Red");
//useState to Show Hide text
const [hideText,setHide]=useState("React Js Hooks");
const[buttontext,setButtontext]= useState("Hide")

// Function to show and hide text and also button text
function showhide() {
  if(buttontext==="Hide")
  {
    setButtontext("Show");
    setHide("")
  }
  else{
    setButtontext("Hide");
    setHide("React Js Hooks")
  }
};

// function to change text value
function changeName(){
  if(name === "LJ University"){
    setName("Welcome Students")
  }else{
    setName("LJ University")
  }
}

// Function to change color of the text
function changeColor(){
  if(textColor === 'red'){
    setcolor("blue")
  }else{
    setcolor("red")
  }
}

return(
```

```

<div>
  <button onClick={changeName}>Change Text</button>
  <button onDoubleClick={changeColor}>Change Color</button>
  <button onClick = {showhide}> {buttontext}</button>

  <h1 style={{color:textColor}}>{name}</h1>
  <h2>{hideText}</h2>
</div>
)}
export default US1

```

## Example:

Write a program to build React app having a button which changes image by clicking it.

### US2.js

```

import { useState } from 'react';
import img1 from "./img1.png";
import img2 from "./img2.png";
function US2 () {
  const [myImage,setImage]=useState(img1);
  function changeImage () {
    if(myImage === img1){
      setImage(img2)
    }else{
      setImage(img1)
    }
  }
  return (
    <div>
      <img src={myImage} height="200px" width="200px" alt="logo" />
      <button onClick={changeImage}>Change Image</button>
    </div>
  ) }
export default US2

```

## Example:

Write React component having a button and image. By clicking on button, image changes randomly from a given array of images.

### US3.js

```
import {useState} from "react";
import img1 from "./img1.jpg"
import img2 from "./img2.jpg"
import img3 from "./img3.png"
import img4 from "./img4.jpg"
import img5 from "./img5.jpg"

function US3()
{
  const arr = [img1,img2,img3,img4,img5]
  const [myimage,setimage] = useState(arr[0]);
  function changeImage {
    const randomIndex = Math.floor(Math.random() * arr.length);
    setimage(arr[randomIndex]);
  };

  return (
    <div className="App">
      <header className="App-header">
        <h1>Random Image Generator</h1>
        <img src={myimage} alt="Random" width="500" height="500"/>
        <button onClick={changeImage}>Change Image</button>
      </header>
    </div>
  );
}

export default US3
```



## Example:

Create a React component that manages multiple form input fields using a single state object and displays the values in real-time

### US4.js

```
import { useState } from 'react'
function US4() {
  const[data, setdata]=useState({});

  function handleChange(e) {
    const { name, value } = e.target;
    setdata({...data,[name]: value});
  };
  return (
    <div>
      <div><input type="text" name="firstName" onChange={handleChange} placeholder='First Name'/></div>
      <div><input type="text" name="lastName" onChange={handleChange} placeholder='Last Name'/></div>
      <h1>First Name: {data.firstName} Lastname: {data.lastName}</h1>
    </div>
  ) }
export default US4
```

## Example:

Write a react component for todo list.

- Add 1 input field and button and by clicking on button display entered task on the same page.
- Also, add delete button with each added task to delete the task.

### Todo.js

```
import {useState} from 'react'
```

```

function Todo() {
  const[Task, setTask]= useState("");
  const[Todolist, setTodoList]=useState([]);
  function handleChange(event) {
    setTask(event.target.value);
  }
  // To add task
  function addTask(event) {
    setTodoList([...Todolist,Task]);
  };
  //To add delete functionality
  function deleteTask(taskName){
    setTodoList(
      Todolist.filter((task)=>{
        if (task!==taskName){
          return true;
        } else{
          return false;
        }
      })
    )
  }
  return (
    <div>
      <h1> Enter Task </h1>
      <input onChange={handleChange}/>
      <button onClick={addTask}> Add Task </button>

      {Todolist.map((task)=>{
        return(
          <div>
            <h1> {task}</h1>
            <button onClick={() => deleteTask(task)}>Delete</button>
          </div>
        )
      })}
    </div>
  )
}

```

```

    );
  }}
</div>
);
}
export default Todo

```

## Example:

Create react app which takes user defined inputs number 1 and number 2 and perform addition, subtraction, multiplication, division of the numbers. (Use useState hook)

US5.js

```

import { useState } from 'react'
function US5() {
  const[data,setData]=useState({});
  const[result,setresult]=useState();

  const handleChange = (e) => {
    const { name, value } = e.target;
    setData({...data,[name]: value});
  };

  function addition(){
    setresult(parseInt(data.num1) + parseInt(data.num2))
  }
  function sub(){
    setresult(parseInt(data.num1) - parseInt(data.num2))
  }
  function mult(){
    setresult(parseInt(data.num1) * parseInt(data.num2))
  }
  function division(){
    setresult(parseInt(data.num1) / parseInt(data.num2))
  }
}

```

```
    return (  
      <div>  
        <div><input type="number" name="num1" onChange={handleChange}  
placeholder='First Name' /></div>  
        <div><input type="number" name="num2" onChange={handleChange}  
placeholder='Last Name' /></div>  
        <button onClick={addition}>addition</button>  
        <button onClick={sub}>Subtraction</button>  
        <button onClick={mult}>Multiplication</button>  
        <button onClick={division}>Division</button>  
        <h1> {result}</h1>  
      </div>  
    ) }  
export default US5
```

## 2. useReducer

The useReducer Hook is the better alternative to the useState hook and is generally more preferred over the useState hook when you have complex state-building logic or when the next state value depends upon its previous value or when the components are needed to be optimized.

You can add a reducer to your component using the useReducer hook. Import the useReducer method from the library like this:

```
import { useReducer } from 'react'
```

The useReducer method gives you a state variable and a dispatch method to make state changes. You can define state in the following way:

```
const [state, dispatch] = useReducer(reducerFunction, initialValue)
```

The reducer function contains your state logic. You can choose which state logic to call using the dispatch function. The state can also have some initial value similar to the useState hook.

Let's understand using example

**The useReducer(reducer, initialState) hook accepts 2 arguments: the reducer function and the initial state. The hook then returns an array of 2 items: the current state and the dispatch function.**

UR.js

```
import React, { useReducer } from 'react';
const initialState = 0;
function reducer(state, action){
  if(action.type==='increment'){
    return state+1;
  }
}
function UR() {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
```

```

<button onClick={() => dispatch({type:"increment"})}>
  Click me ({state})
</button>
);
}
export default UR

```

### A. Initial state

The initial state is the value the state is initialized with. Here initialized with 0.

### B. Reducer function

The reducer is a pure function that accepts 2 parameters: **the current state and an action object**. Depending on the action object, the reducer function must update the state in an immutable manner, and return the new state.

### C. Action object

An action object is an object that describes how to update the state.

Typically, the action object has a property **type** — a string describing what kind of state update the reducer must do.

If the action object must carry some useful information to be used by the reducer, then you can add additional properties to the action object. **Here we have defined type “increment”.**

### D. Dispatch function

The dispatch is a special function that dispatches an action object.

The dispatch function is created for you by the useReducer() hook:

```
const [state, dispatch] = useReducer(reducer, initialState);
```

Whenever you want to update the state (usually from an event handler or after completing a fetch request), you simply call the dispatch function with the appropriate action object: dispatch(actionObject).

Like as below in above example

```
<button onClick={() => dispatch({type:"increment"})}>
```

**In simpler terms, dispatching means a request to update the state.**

[What Is a Reducer and Why do You Need It?](#)

Let's take an example of a To-Do app. This app involves adding, deleting, and updating items in the todo list. The update operation itself may involve updating the item or marking it as complete.

When you implement a todo list, you'll have a state variable `todoList` and make state updates to perform each operation. However, these state updates may appear at different places, sometimes not even inside the component.

To make your code more readable, you can move all your state updates into a single function that can exist outside your component. While performing the required operations, your component just has to call a single method and select the operation it wants to perform.

The function which contains all your state updates is called the reducer. This is because you are reducing the state logic into a separate function. The method you call to perform the operations is the dispatch method.

## Example:

**Write react component to increase value by 5 while clicking on button. Initialize value with 20. Use `useReducer` hook to perform the task.**

### UR1.js

```
import React, { useReducer } from 'react'
function reducer(state,action){
  return state+action;
}
function UR1 () {
  const [state,dispatch]=useReducer(reducer ,20);
  return (
    <div align="center">
      <h1 align="center">{state}</h1>
      <button onClick={()=>dispatch(5)}>Add</button>
    </div>
  )
}
export default UR1
```

## Example:

Create react js app to increase value by 1 while clicking on button “Increment” and decrease value by 1 while clicking on button “Decrement”. Initialize value with 0. Use useReducer hook to perform the task.

### Usereducer.js

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

const initialState=0;
function reducer(state,action){
  if(action.type==='increment'){
    return state+1;
  }
  if(action.type==='decrement'){
    return state-1;
  }
}
function Usereducer(){
  const[state,dispatch] = useReducer(reducer,initialState);
  return(
    <>
      <h1>{state}</h1>
      <button onClick={()=> dispatch({type:"increment"})}> Increment </button>
      <button onClick={()=>dispatch({type:"decrement"})}> Decrement </button>
    </>
  )
}
export default Usereducer
```



### 3.useContext

Context provides a way to pass data or state through the component tree without having to pass props down manually through each nested component. It is designed to share data that can be considered as global data for a tree of React components.

#### How to use the context

Using the context in React requires 3 simple steps:

- creating the context,
- providing the context,
- consuming the context.

#### A. Creating the context

The built-in function `createContext(default)` creates a context instance:

```
import { createContext } from 'react';
const Fname = createContext('Default Value'); //default value is optional
```

The function accepts one optional argument: the default value.

#### B. Providing the context

**Context.Provider** component available on the context instance is used to provide the context to its child components, no matter how deep they are.

To set the value of context use the **value prop**

```
< Fname.Provider value="Test" />
```

#### Main.js

```
import React,{ createContext } from 'react';
import Comp from './Comp'
const Fname = createContext();
function Main() {
  return (
    < Fname.Provider value='ABC'>
      < Comp /> // component in which consuming the context
    </ Fname.Provider>
  );
}
export default Main
export {Fname}
```

Again, what's important here is that all the components that'd like later to consume the context have to be wrapped inside the provider component.

If you want to change the context value, simply update the value prop.

### C. Consuming the context

Consuming the context can be performed by the `useContext(Context)` React hook:

#### Comp.js

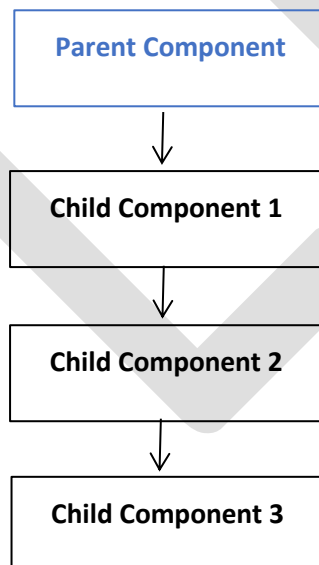
```
Import React,{ useContext } from 'react';  
import { Fname } from './Main;
```

```
function Comp () {  
  const value = useContext(Fname);  
  return <span>{value}</span>;  
}
```

Export default **Comp**

The hook returns the value of the context: `value = useContext(Context)`. The hook also makes sure to re-render the component when the context value changes.

Suppose, we have one parent component and we want to access its data in child component 3. Then, we can use `createContext` for creating a context and `usecontext` to use the context.



## Example:

Write a reactJS program to perform the tasks as asked below.

- Create one main file (parent file) name PC.js and other 2 component files C1.js and C2.js
- Pass First name and Last name from PC.js file to C2.js file. And display **Welcome ABC XYZ** (suppose firstname is ABC and Last name is XYZ) in browser.

### PC.js

```
import React, { createContext } from "react"
import C1 from "./C1"
// To create context for firstname and lastname
const Fname = createContext();
const Lname = createContext();

function PC(){
  return (
    <>
      <Fname.Provider value="ABC"> // to provide the value which should be passed
        <Lname.Provider value="XYZ">
          <C1/>
        </Lname.Provider>
      </Fname.Provider>
    </>
  )
}
export default PC
export {Fname,Lname} //Each component must have one default export, so the context should be
exported using {} as object.
```

### C1.js

```
import React from "react"
import C2 from "./C2"
function C1(){
  return ( <C2/> )
}
export default C1
```

**C2.js**

```
import React, { useContext } from "react"
import { Fname,Lname } from "./PC"
function C2(){
  const fn = useContext(Fname)
  const ln = useContext(Lname)
  return (
    <h1>Welcome {fn} {ln}</h1>
  )
}
export default C2
```

**App.js** <- **PC.js** <- **C1.js** <- **C2.js**

So, **First Name** and **Last Name** are defined in **PC.js** component and are used **directly in C2.js component** without passing data to all other hierarchical components.

**Example:**

Write a reactJS program to perform the tasks as asked below.

- Create one main file (parent file) name **Comp.js** and other 3 component files **Comp1.js**, **Comp2.js**, **Comp3.js**.
- Pass **Number1** and **Number 2** from **Comp.js** file to **Comp3.js** file. Calculate multiplication of the numbers using **useContext**.

**Comp.js**

```
import React, { createContext } from "react"
import Comp1 from "./Comp1"
const Num1 = createContext();
const Num2 = createContext();
function Comp(){
  return (
    <>
      <Num1.Provider value="20">
        <Num2.Provider value="5">
          <Comp1/>
        </Num2.Provider>
      </Num1.Provider>
    </>
  )
}
```

```

        </Num2.Provider>
    </Num1.Provider>
</>
)
}
export default Comp
export {Num1,Num2}

```

### Comp1.js

```

import React from "react"
import Comp2 from "../Comp2"

function Comp1(){
    return ( <Comp2/> )
}
export default Comp1

```

### Comp2.js

```

import React from "react"
import Comp3 from "../Comp3"
function Comp2(){
    return (
        <Comp3/>
    )
}
export default Comp2

```

### Comp3.js

```

import React, { useContext } from "react"
import { Num1,Num2 } from "../Comp"

function Comp3(){
    const num1 = useContext(Num1)
    const num2 = useContext(Num2)
    return (
        <h1>Multiplication of numbers in component-3: {num1 * num2}</h1>
    )
}

```

```
export default Comp3
```

**App.js** <- **Comp.js** <- **Comp1.js** <- **Comp2.js** <- **Comp3.js**

So, **Number 1 and number 2 are defined in Comp.js** component and are used **directly in Comp3.js component** without passing data to all other hierarchical components.

### Example:

Use multiple contexts in a React application by creating and consuming them across different components.

**Comp1.js:** Creates a context for CSS styling and provides it to **Comp2**.

**Comp2.js:** Creates a context for a string value ("Students") and provides it to **Comp3**.

**Comp3.js:** Consumes both contexts and displays a message with the provided styles and string.

#### Comp1.js

```
import { createContext } from "react"
import Comp2 from "./Comp2"
const CC = createContext();
const mycss={backgroundColor:'yellow',color:'red',fontSize:"45px"}
function Comp(){
  return (
    <>
      <CC.Provider value={mycss}>
        <Comp2/>
      </CC.Provider>
    </>
  )
}
export default Comp
export {CC}
```

#### Comp2.js

```
import { createContext } from "react"
import Comp3 from "./Comp3"
const CC1 = createContext();
```

```
function Comp(){  
  return (  
    <CC1.Provider value="Students">  
      <Comp3/>  
    </CC1.Provider>  
  )  
}  
export default Comp  
export {CC1}
```

### Comp3.js

```
import { useContext } from "react"  
import { CC } from "../Comp1"  
import { CC1 } from "../Comp2"  
function Comp3(){  
  const mycss = useContext(CC)  
  const data = useContext(CC1)  
  return (  
    <h1 style={mycss}>Welcome to useContext tutorial {data}</h1>  
  )  
}  
export default Comp3
```

## 4. useEffect

The useEffect Hook allows you to perform side effects in your components.

Some examples of side effects are: fetching data, directly updating the DOM, and timers.

useEffect accepts two arguments. The second argument is optional.

```
useEffect(<function>, <dependency>)
```

### To import useEffect

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

Below is the example to understand concept of empty array and array with value of useEffect

Write react js app to perform the tasks as asked below.

- Add two buttons and increment count by one with each click.
- Display alert as an effect on specified conditions.
  - Effect will be triggered only when page rendered for the 1<sup>st</sup> time. (empty array)
  - Effect will be triggered every time the button A is clicked. (array with value)
  - When the Page is Rendered for the First Time and On Every Update/Event Triggered

### UE1.js

```
import {useState,useEffect } from 'react'
function UE1() {
  const[count,setcount]=useState(0);
  const[calulation,setcal]=useState(0);
  // when the page is rendered for first time and also when Button A(count) clicked.
  useEffect(()=>
  {
    alert("Clicked")
  },[count]);
  // only once when the page is rendered
  useEffect(()=>
  {
    alert("Clicked")
  },[]);
```



// when the page is rendered for first time and also on every update/event triggered

```
useEffect(()=>
{
  alert("Clicked")
});

const changeCount={()=>{
  setcount(count+1);
}}
const changeCalc={()=>{
  setcal(calculation+1);
}}
return (
  <div>
    <button onClick={changeCount}>Button A {count}</button><br/>
    <button onClick={changeCalc}>Button B {calculation}</button>
  </div>
) }
export default UE1
```

## Explanation of useEffect Usage

**When the Page is Rendered for the First Time and When Button A (Count) is Clicked:**

```
useEffect(() => {
  alert("Clicked");
}, [count]);
```

This useEffect runs whenever the count state changes, including the initial render. So, every time Button A is clicked and count is updated, the effect runs, showing the alert "Clicked".

**When the Page is Rendered for the First Time Only:**

```
useEffect(() => {
  alert("Clicked");
}, []);
```

This useEffect with an empty dependency array runs only once when the component mounts. It won't run again unless the component is unmounted and remounted.

**When the Page is Rendered for the First Time and On Every Update/Event Triggered:**

```
useEffect(() => {  
  alert("Clicked");  
});
```

This `useEffect` runs on every render, including the initial render and any subsequent updates. This is because there is no dependency array, so it runs every time the component renders.

If your alert is showing twice initially, it is likely because of React's Strict Mode. In development mode, React's Strict Mode intentionally invokes certain lifecycle methods (like `useEffect`) twice to help identify potential issues.

### Explanation

React Strict Mode is a tool for highlighting potential problems in an application. It does so by intentionally invoking certain methods, such as component mounts and updates, twice. This behavior is designed to help developers find bugs related to side effects that might be dependent on certain lifecycle methods.

### How to Check if Strict Mode is the Cause

You can check if this is the issue by looking at your `index.js` file where the React application is initialized. If you see `React.StrictMode` wrapping your application, that's the reason for the double invocation.

### Write a ReactJS script to create a digital clock running continuously. (useEffect) UE2.js

```
import { useState, useEffect } from 'react';  
  
function UE2() {  
  const [date, setDate] = useState(new Date());  
  
  useEffect(() => {  
    const timer = setInterval(() => {  
      setDate(new Date());  
    }, 1000);  
    return () => {  
      clearInterval(timer);  
    };  
  }, []);  
}
```

```
return (  
  <h1>  
    Time using LocalTimeString - {date.toLocaleTimeString()}<br />  
    Hour-{date.getHours()}:Min-{date.getMinutes()}:Sec-{date.getSeconds()}  
  </h1>  
);  
}  
  
export default UE2;
```

## Explanation:

- ✓ `useEffect(..., [])` ensures the interval is set **only once** when the component mounts.
- ✓ `setInterval` updates the time every second.

## Without `useEffect` Issues


- ✓ Every time the component re-renders, a new `setInterval` is created.
- ✓ You'll end up with **multiple intervals**, causing performance issues and rapid updates.

## React Forms

- ✓ Forms are an integral part of any modern web application. It allows the users to interact with the application as well as gather information from the users.
- ✓ Forms can perform many tasks that depend on the nature of your business requirements and logic such as authentication of the user, adding user, searching, filtering, booking, ordering, etc. A form can contain text fields, buttons, checkbox, radio button, etc.
- ✓ In React, form data is usually handled by the components. When the data is handled by the components, all the data is stored in the component state.
- ✓ You can control changes by adding event handlers in the `onChange` attribute and that event handler will be used to update the state of the variable.

 **Adding Forms in React** You add a form with React like any other element:

```
function MyForm()  
{ return(  
  <form>  
    <label>Enter your name:  
    <input type="text" /> </label>  
  </form>  
) }
```

 **Handling Forms** Handling forms is about how you handle the data when it changes value or gets submitted.

In React, form data is usually handled by the components. When the data is handled by the components, all the data is stored in the component state.

You can control changes by adding event handlers in the `onChange` attribute. We can use the `useState` Hook to keep track of each inputs.

```
import { useState } from 'react';  
function MyForm() {  
  const [name, setName] = useState("");  
  return (  
    <form>  
      <label>Enter your name:  
      <input type="text" value={name} onChange={(e) => setName(e.target.value)} />  
    </label>  
    </form>  
  )  
}
```

```
)
}
```

### ✚ Submitting Forms

You can control the submit action by adding an event handler in the **onSubmit** attribute for the **<form>**:

```
import { useState } from 'react';
function MyForm() {
  const [name, setName] = useState("");
  function handleSubmit (event) {
    event.preventDefault();
    alert(`The name you entered was: ${name}`)
  }
  return (
    <form onSubmit={handleSubmit}>
    <label>Enter your name:
    <input type="text" value={name} onChange={(e) => setName(e.target.value)} />
    </label>
    <input type="submit" />
    </form>
  )
}
```

## Form Fields

### ❖ Textarea

The textarea element in React is slightly different from ordinary HTML. In React the value of a textarea is placed in a value attribute. We'll use the **useState** Hook to manage the value of the textarea:

```
import { useState } from 'react';
function MyForm() {
  const [txtarea, setTxtarea] = useState(
    "The content of a textarea goes in the value attribute"
  );
  function handleChange(event) {
    setTxtarea(event.target.value)
  }
}
```

```

return (
<form>
<textarea value={txtarea} onChange={handleChange} />
</form>
)
}

```

### ❖ Select

A drop down list or a select box, in React is also a bit different from HTML. In React, the selected value is defined with a **value** attribute on the **select** tag:

```

function MyForm() {
const [myCar, setMyCar] = useState("Volvo");
function handleChange(event) {
setMyCar(event.target.value)
}
return (
<form>
<select value={myCar} onChange={handleChange}>
<option value="Ford">Ford</option>
<option value="Volvo">Volvo</option>
<option value="Fiat">Fiat</option>
</select>
</form>
)
}

```

### ❖ Radio Button

**Example:- Create a React Form to select any of pizza size using radio button.**

```

import React from 'react';
import { useState } from 'react';
function Myform() {
  const [s, setSize] = useState('Medium');

  function onOptionChange(e) {
    setSize(e.target.value);
  };
}

```

```
return (  
  <div>  
    <h3>Select Pizza Size</h3>  
    <form>  
      <input type='radio' name='ps' value='Regular' checked={s === 'Regular'}  
        onChange={onOptionChange} />  
      <label>Regular</label>  
  
      <input type='radio' name='ps' value='Medium' checked={s === 'Medium'}  
        onChange={onOptionChange} />  
      <label>Medium</label>  
  
      <input type='radio' name='ps' value='Large' checked={s === 'Large'}  
        onChange={onOptionChange} />  
      <label>Large</label>  
    </form>  
    <p>  
      Selected size <strong>{s}</strong>  
    </p>  
  </div>  
);}  
export default Myform
```

## Example

Create react app which contains form with following fields.

- First Name(Input type text)
- Email(Input type email)
- Password(Input type password)
- Confirm Password(Input type password)
- Message (Textarea)
- Gender(Radio Button)
- City (Dropdown)

Display submitted values in alert box. (Using useState Hook)

### Form1.js

```
import { useState } from 'react'

function Form1() {
  const [formdata, setformdata] = useState({});
  function handleChange(event) {
    const {name, value} = event.target;
    setformdata({...formdata, [name]: value})
  }
  function handlesubmit(e){
    e.preventDefault();
    alert("Your form has been submitted.\nName: " + formdata.fname + "\nEmail: " +
formdata.eid + "\nCity: " + formdata.city + "\nGender: " + formdata.gender)
  }
  return (
    <div>
      <form onSubmit={handlesubmit}>

        <label>First Name:</label>
        <input type="text" name="fname" onChange={handleChange} /><br/>

        <label>Email Id:</label>
        <input type="email" name="eid" onChange={handleChange} /><br/>
```



```

<label>Password:</label>
<input type="password" name="pass" onChange={handleChange} required/><br/>

<label>Confirm Password:</label>
<input type="password" name="cpass" onChange={handleChange} /><br/>

<label>Message :</label>
<textarea name="msg" onChange={handleChange} /><br/>

<select onChange={handleChange} name='city'>
  <option value="Ahmedabad">Ahmedabad</option>
  <option value="Rajkot">Rajkot</option>
</select>

<input type="radio" name="gender" value="Male" onChange={handleChange} />Male
<input type="radio" name="gender" value="Female"
onChange={handleChange}/>Female

<button type="submit">Submit</button> <br/>
</form>
</div>
)
}
export default Form1

```

## Example

Create react app which contains form with following fields.

- Email(Input type email)
- Password(Input type password)
- Confirm Password(Input type password)
- Add validation using regex to validate email id and password (Must contain at least 8 characters and must contain at least 1 uppercase, 1 lowercase and 1 digit).
- Also values of password and confirm password must be same.

Display email in alert box. (Using useState Hook)

### Form2.js

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

function Form2() {
  const [formdata, setformdata] = useState({});
  function handleChange(event) {
    const { name, value } = event.target;
    setformdata({ ...formdata, [name]: value })
  }
  function handleSubmit(e)
  {
    e.preventDefault();
    const emailregex = /^[a-zA-Z0-9._-]+@[a-zA-Z]+\.[a-zA-Z]{2,4}$/;
    const passregex = /^(?=.*[0-9])(?=.*[a-z])(?=.*[A-Z]).{8,}/;
    if (!emailregex.test(formdata.eid))
    {
      alert("Please enter a valid Email ID");
    }
    else if (!passregex.test(formdata.pass))
    {
      alert("Password must contain atleast 8 characters ( Atleast 1 digit, 1 lowercase & 1 uppercase alphabets )");
    }
    else if (formdata.pass !== formdata.cpass) {
      alert("password and confirm password must be same");
    }
  }
}
```

```

    }
    else{
      alert("Your form has been submitted.\nEmail: " + formdata.eid )
    }
  }
  return (
    <div>
      <form onSubmit={handlesubmit}>

        <label>Email Id:</label>
        <input type="email" name="eid" onChange={handlechange} /><br/>

        <label>Password:</label>
        <input type="password" name="pass" onChange={handlechange} /><br/>

        <label>Confirm Password:</label>
        <input type="password" name="cpass" onChange={handlechange} /><br/>

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

    </div>
  )
}
export default Form2

```

## Example

Create react app which contains form with fields Name, Email Id, Password and Confirm Password. When the form submitted the values of password and confirm password fields must be same else it will give an error message in alert box. If form submitted successfully then display entered name and email id in alert box.

### Form2.js

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

```
function Form2(){
  const[formdata,setformdata]=useState({});
  const handlechange = (event) => {
    const name = event.target.name;
    const value = event.target.value;
    setformdata({...formdata, [name]: value})
  }
  const handlesubmit=(e)=>
  {
    e.preventDefault();
    if(formdata.pass !== formdata.cpass){
      alert("Values of Password and Confirm password must be same")
    }else{
      alert("Welcome "+formdata.fname+"\n Your Email id is: "+ formdata.eid)
    }
  }
  return (
    <div>
      <form className="form-data" onSubmit={handlesubmit}>

        <label>Name:</label>
        <input type="text" name="fname" onChange={handlechange} /><br/>
        <label>Email Id:</label>
        <input type="email" name="eid" onChange={handlechange} required/><br/>
        <label>Password :</label>
        <input type="password" name="pass" onChange={handlechange} required/><br/>
        <label>Confirm Password :</label>
        <input type="password" name="cpass" onChange={handlechange} /><br/>
        <button type="submit">Submit</button> <br/>
      </form>

    </div>
  )
}
export default Form2
```

## Axios

### What is Axios?

- ✓ It is a library which is used to make requests to an API, return data from the API, and then do things with that data in our React application.
- ✓ **Axios** is a popular **JavaScript library** used to make **HTTP requests** (like GET, POST, PUT, DELETE) from the browser or Node.js.

### Syntax:

```
const getPostsData = () => {  
  axios  
    .get("API URL")  
    .then(data => console.log(data.data))  
    .catch(error => console.log(error));  
};  
getPostsData();
```

### Installing Axios

- In order to use Axios with React, we need to install Axios. It does not come as a native JavaScript API, so that's why we have to manually import into our project.
- Open up a new terminal window, move to your project's root directory, and run any of the following commands to add Axios to your project.

```
npm install axios
```

To perform this request when the component mounts, we use 'useEffect' hook.

This involves

- importing Axios (**import axios from "axios";**)
- using the **.get()** method to make a GET request to your endpoint,
- using a **.then()** callback to get back all of the response data.

What if there's an error while making a request? For example, you might pass along the wrong data, make a request to the wrong endpoint, or have a network error. In this case, instead of executing the **.then()** callback, Axios will throw an error and run the **.catch()** callback function. In this function, we are taking the error data and putting it in state to alert our user about the error. So if we have an error, we will display that error message.

## Example

Create a react app to display images by requesting API using Axios.

```
import React,{ useState,useEffect } from 'react'
import axios from "axios";
const Randomimage = () =>
{
  const[myimg,setimg]=useState("");

  useEffect(() => {
    setInterval(() => {
      axios
      .get('https://dog.ceo/api/breeds/image/random')
      .then((response)=>{console.log(response.data);setimg(response.data);})
      .catch((error)=>{console.error(error);})
    }, 2000)
  },[])

  return(
    <div>
      <img src={myimg.message} alt='image' height={300} width={300}/>
    </div>
  )
}
export default Randomimage
```

- **useEffect**: This hook runs side effects in functional components.
- The empty dependency array `[]` means the effect runs only once after the initial render.
- **setInterval**: Sets up a function to run every 2 seconds (2000 milliseconds).
- **axios.get('https://dog.ceo/api/breeds/image/random')**: Makes a GET request to the Dog CEO API to fetch a random dog image.
- **then()**: Executes when the API call is successful. The response data, which contains the image URL, is logged to the console and stored in the `myimg` state.
- **catch()**: Executes if there's an error with the API call, logging the error to the console.

## Example

Write React component that fetches a random joke from the API and displays it when the "Generate Joke" button is clicked.

```
import React,{ useState,useEffect } from 'react'
import axios from "axios";
const Randomjokeapi = () =>
{
  const[joke,setJoke]=useState("");
  function fetchJoke() {
    axios
    .get("https://official-joke-api.appspot.com/random_joke")
    .then((response)=>{setJoke(response.data);})
    .catch((error)=>{console.error(error);})
  }
  useEffect(fetchJoke,[])
  return(
    <div>
      <h1>{joke.setup}</h1>
      <h3>{joke.punchline}</h3>
      <button onClick={fetchJoke} >Generate Joke </button>
    </div>
  )
}
export default Randomjokeapi
```

[https://official-joke-api.appspot.com/random\\_joke](https://official-joke-api.appspot.com/random_joke) link contains object as shown below. We have used setup and punchline properties for our webpage.

```
{"type":"general","setup":"What has ears but cannot hear?","punchline":"A field of corn.","id":238}
```

- ✓ **useEffect(fetchJoke, [])** : This fetches a joke once when the component loads and lets the user fetch more with the button
- ✓ **joke**: to store the joke data, specifically the setup and punchline properties.
- ✓ fetchjoke uses axios.get to fetch a random joke from the API.
- ✓ On success, updates the joke state with the setup and punchline from the fetched data.
- ✓ On failure, logs the error to the console and updates the error state with a user-friendly message.