**React**

React is an open source JS library for building user interface.

* React is not a framework.
* It only focuses on UI.
* Created and maintained by FB.
* It has component based architecture.
* React is declarative - **we write the code that we want and React is in charge of taking our declared code and performing all of the JavaScript/DOM steps to get us to our desired result**. (The opposite of declarative is imperative. The imperative approach is when you provide step-by-step DOM mutations until you reach desired UI.)
* Easy to integrate into any application.

Install react project

npx create-react-app <hello-world>

npm start

npx is a npm package runner which gets installed when you installed node and that is how we are directly able to run create-react-app without having to install it. Actually npx takes care that for us.

[We can also install the same react project using npm-

npm install create-react-app -g Install the create-react-app package globally

create-react-app <project-name> Then use the package to generate the projects

Here you have to update constantly the create-react-app package. ]

Components

Types-

1. Stateless functional component – JS functions that return HTML which describes the UI.

2. Stateful class component – class extending component class. Render method returning HTML.

React JSX

JSX stands for JavaScript XML. JSX allows us to write HTML in React. JSX makes it easier to write and add HTML in React.

JSX allows us to write HTML elements in JavaScript and place them in the DOM without any createElement()  and/or appendChild() methods. JSX converts HTML tags into react elements.

State in React

A state is nothing but an object that is privately maintained inside a component. A state can influence what is rendered in the browser and lastly state can be changed within the component.

setState in React

Always make use of setState never modify the state directly.

Code has to be executed after the state has been updated? Place the code in the call back function which is the second argument to the setState method.

increment() {

    this.setState(

      {

        count: this.state.count + 1,

      },

      () => {

        console.log("callback value", this.state.count);

      }

    );

    console.log(this.state.count);

  }

When you have to update state based on the previous state value, pass in a function as an argument instead of the regular object.

increment() {

    this.setState((prevState) => ({ count: prevState.count + 1 }));

    console.log(this.state.count);

  }

Context

Context provides a way to pass data through the component tree without having to pass props down manually at every level.

Steps:

1. Create the context

userContext.js

import React from "react";

const UserContext = React.createContext();

const UserProvider = UserContext.UserProvider;

const UserConsumer = UserContext.Consumer;

export { UserProvider, UserConsumer };

1. Provide a context value
2. Consume the context value

HTTP and React

React is a library for building user interfaces. It is in no way concerned about HTTP. This raises a very important question how do we make ajax request in react or how do we make API calls in react. Actually, react itself does not have any particular way to fetch or send data to the server. In fact, react even have to know that there is a server in the picture. React components simply reads the props and state and render the UI. Therefore, to use some data from the server you just have to get the data into your components, props or state.

So, if react isn’t going to handle the request, then who is doing this? This is where you have to use HTTP library. There are few popular ones but we will use Axios. It’s a promise based library. By this command you can add Axios package in your application:

npm install axios

When the command is completed, Axios should be included in the list of dependencies in package.json file.

*Get request (PostList.js)*

import React, { Component } from "react";

import axios from "axios"; /\* 1. Include axios\*/

class PostList extends Component {

  constructor(props) {

    super(props);

    this.state = {

posts: [], /\* 2. In the constructor we need to create a state property which will store the list of posts

So, we have a state property called posts initialized to an empty array \*/

    };

  }

  componentDidMount() {

    axios

      .get("https://jsonplaceholder.typicode.com/posts") /\* 3. Use axios to make a get request to the

      .then((response) => { jesonplaceholder API \*/

        //console.log(response);

        this.setState({ posts: response.data });

      })

      .catch((error) => {

        //console.log(error);

this.setState({ errorMsg: "Error retreiving data" });

      });

  }

  render() {

    const { posts,errorMsg } = this.state; /\* 4. Destructure the state property \*/

    //console.log(posts);

    return (

      <div>

        <h1>PostList</h1>

        {posts.length

          ? posts.map((post) => <div key={post.id}>{post.title}</div>)

          : null} /\* 5. Displays the list of post using map method \*/

       {errorMsg ? <div>{errorMsg}</div> : null}

</div>

    );

  }

}

export default PostList;

*Post request (PostForm.js)*

import React, { Component } from "react";

import axios from "axios"; /\* 6. Include axios\*/

class PostForm extends Component {

  constructor(props) {

    super(props);

    this.state = { {/\* 2. create state properties for the fields and link them back to the input elements name \*/}

      userId: "",

      title: "",

      body: "",

    };

  }

{/\* 4. Changehandler to track the changes and insync with the state object \*/}

changeHandler = (e) => {

    this.setState({ [e.target.name]: e.target.value });

  };

{/\* 5. submitHandler \*/}

 submitHandler = (e) => {

    e.preventDefault();

    //console.log(this.state);

    axios

      .post("https://jsonplaceholder.typicode.com/posts", this.state)

      .then((response) => {

        console.log(response);

      })

      .catch((error) => {

        console.log(error);

      });

  };

  render() {

{/\* 3. destructure them in the render method and add their values to the value attribute of inputs elements\*/ }

const { userId, title, body } = this.state;

    return (

      <div>

        <form onSubmit={this.submitHandler}>    {/\* 1. create a form  \*/}

         <div>

            <input type="text" name="userId" value={userId}  onChange={this.changeHandler}

/>

          </div>

          <div>

            <input type="text" name="title" value={title}  onChange={this.changeHandler}

/>

          </div>

          <div>

            <input type="text" name="body" value={body onChange={this.changeHandler}

/>

          </div>

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

        </form>

      </div>

    );

  }

}

export default PostForm;

<https://www.youtube.com/watch?v=-ZMP8ZladIQ>

Hooks and React

* Hooks are a new feature addition in React v16.8
* They allow you to use React features without having to write a class
* Avoid the whole confusion with ‘this’ keyword. Since hooks don’t have class components also minified better.
* It allow you to reuse Stateful logic without changing component hierarchy
* Organize the logic inside a component into reusable isolated units. Mutually related code can be put together which will avoid trivial bugs and inconsistencies.
* Organize the logic inside a component into reusable isolated units. Mutually related code can be put together which will avoid trivial bugs and inconsistencies.

\*\* Rules of Hooks

* “Only call hooks at the top level” – don’t call hooks inside loops, conditions or nested functions.
* “Only call hooks from react functions” – call them from within react functional components and not just any regular JavaScript function.

**useState Hook**

1. The useState hook let you add state to functional components.
2. In classes, the state is always an object.
3. With the useState hook, the state doesn’t have to be an object.
4. The useState hook returns an array with 2 elements.
5. The first element is the current value of the state, and the second element is a state setter function.
6. New state value depends on the previous state value. You can pass a function to the setter function
7. When dealing with objects or arrays, always make sure to spread your state variable and then call the setter function.

Hooks are just functions so we simply call the useState. This hook accepts argument – initial value of state property and returns the current value of the state property and a method that is capable of updating that state property.

HookCount.js

import React, { useState } from "react";

export default function HookCounter() {

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

  return (

    <div>

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

    </div>

  );

}

// **\*\*** state variable count initialized to 0 and a method setCount capable of changing the count variable

# useState with previous state

HookCountTwo.js

import React, { useState } from "react";

export default function HookCountTwo() {

  const initialCount = 0;

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

  const incrementFive = () => {

    for (let i = 0; i < 5; i++) {

      setCount((prevCount) => prevCount + 1);

    }

  };

  return (

    <div>

      <div>Count : {count}</div>

      <button onClick={() => setCount(initialCount)}>Reset</button>

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

        Increment

      </button>

      <button onClick={() => setCount((prevCount) => prevCount - 1)}>

        Decrement

      </button>

      <button onClick={incrementFive}>Increment 5</button>

    </div>

  );

}

# useState with object

1. This time default value of the state variable is an object which is initialized to an empty string. A state variable can be Boolean, object, number or array.
2. When user typing in something in the firstname , we want to set the firstname property. So event(e) arrow function call set name which is the setter function for the name state variable. And what do we want to set? FirstName equal to event.target.value.
3. …name - will copy every property in the object and then just overwrite the first name field with a different value. Actually, setter function provided by the useSatet hook doesn’t automatically merge and update objects. We have to merge it manually and then pass the value to the setter function.

import React, { useState } from "react";

export default function HookCounterThree() {

  const [name, setName] = useState({ firstName: "", lastName: "" });

**1**

  return (

    <div>

      <h1>useState with object</h1>

      <form>

        <div>

          <label>FirstName </label>

          <input

            type="text"

**3**

**2**

            value={name.firstName}

            onChange={(e) => setName({ ...name, firstName: e.target.value })}

          ></input>

        </div>

        <div>

          <label>LatName </label>

          <input

            type="text"

            value={name.lastName}

            onChange={(e) => setName({ ...name, lastName: e.target.value })}

          ></input>

        </div>

        <div>

          <h2>FirstName: {name.firstName}</h2>

          <h2>LastName: {name.lastName}</h2>

        </div>

        {JSON.stringify(name)}

      </form>

    </div>

  );

}

# useState with array

1. …items - will copy every property in the array and then just overwrite the items with a different value. Actually, setter function provided by the useSatet hook doesn’t automatically merge and update the array. We have to merge it manually and then pass the value to the setter function.

import React, { useState } from "react";

function HookCounterFour() {

  const [items, setItems] = useState([]);

  const addItem = () => {

    setItems([...items, { id: items.length, value: "react" + items.length }]);

  };

**1**

  return (

    <div>

      <button onClick={addItem}> Add Item</button>

      <ul>

        {items.map((item) => (

          <li key={item.id}>{item.value}</li>

        ))}

      </ul>

    </div>

  );

}

export default HookCounterFour;

**useEffetc Hook**

useEffect hook is a close replacement for componentDidMount, componentDidUpdate and componenetWillUnmount.

\* useEffect hook runs after every single render*.*

Fetching data

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

import axios from "axios";

export default function DataFetching() {

  const [posts, setPosts] = useState([]);

  useEffect(() => {

    axios

      .get("https://jsonplaceholder.typicode.com/posts")

      .then((res) => setPosts(res.data))

      .catch((error) => console.log(error));

  });

  return (

    <div>

      <ul>

        {posts.map((post) => (

          <li key="{post.id}">{post.title}</li>

        ))}

      </ul>

    </div>

  );

}

Fetching particular data

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

import axios from "axios";

export default function DataFetching() {

  const [post, setPost] = useState({});

  const [id, setId] = useState(1);

  useEffect(() => {

    axios

      .get(`https://jsonplaceholder.typicode.com/posts/${id}`)

      .then((res) => setPost(res.data))

      .catch((error) => console.log(error));

  }, [id]);

  return (

    <div>

      <input type="text" value={id} onChange={(e) => setId(e.target.value)} />

      <div>{post.title}</div>

    </div>

  );

}

Fetching particular data from a button click

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

import axios from "axios";

export default function DataFetching() {

  const [post, setPost] = useState({});

  const [id, setId] = useState(1);

  const [idFormButtonClick, setIdFormButtonClick] = useState(1);

  useEffect(() => {

    axios

      .get(`https://jsonplaceholder.typicode.com/posts/${idFormButtonClick}`)

      .then((res) => setPost(res.data))

      .catch((error) => console.log(error));

  }, [idFormButtonClick]);

  const handelClick = () => {

    setIdFormButtonClick(id);

  };

  return (

    <div>

      <input type="text" value={id} onChange={(e) => setId(e.target.value)} />

      <button type="button" onClick={handelClick}>

        Fetch Post

      </button>

      <div>{post.title}</div>

    </div>

  );

}

**useContext**

Context provides a way to pass data through the component tree without having to pass props down manually at every level.

App.js

import React from "react";

import "./App.css";

import ComponentC from "./components/ComponentC";

export const UserContext = React.createContext();

**1**

export const ChannelContext = React.createContext();

function App() {

  return (

    <div className="App">

      <UserContext.Provider value={"Viswas"}>

**2**

        <ChannelContext.Provider value={"Codevolution"}>

          <ComponentC />

        </ChannelContext.Provider>

      </UserContext.Provider>

    </div>

  );

}

export default App;

ComponentC

import React from "react";

import ComponentE from "./ComponentE";

function ComponentC() {

  return (

    <div>

      <ComponentE />

    </div>

  );

}

export default ComponentC;

ComponentE

**3**

import React, { useContext } from "react";

**4**

import { UserContext, ChannelContext } from "../App";

function ComponentE() {

  const user = useContext(UserContext);

**5**

  const channel = useContext(ChannelContext);

  return (

    <div>

      {user} - {channel}

    </div>

  );

}

export default ComponentE;

1. Create the context
2. Provide this context with the value. Provider must wrap the children component for the value to be available.
3. Import useContext hook from react
4. Import necessary context. Here we have two - UserContext, ChannelContext
5. Finally, call the useContext function passing in the context as its argument. Here useContext returns the context value. Let assign the function call to a variable and use them in JSX to render in the UI.

**userReducer**

useReducer is a hook that is used for state management in react. It is an alternative to useState. Actually, useState is built using useReducer. So useReducer is a primitive hook compared to useState. The useReducer hook accepts two parameters – reducer function and initialState.

useReducer(reducer, initialState)

Now, the reducer function also accepts two parameters – currentState and action.

reducer(currentState, action)

reduce method returns a single value. Whereas useReducer returns a pair of values – [newState, dispatch]

useReducer (simple state & action)

**1**

import React, { useReducer } from "react";

**4**

const initialState = 0;

const reducer = (state, action) => {

  switch (action) {

    case "increment":

      return state + 1;

    case "decrement":

**5**

      return state - 1;

    case "reset":

      return initialState;

    default:

      return state;

  }

};

**2**

function CounterOne() {

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

**6**

  return (

    <div>

**7**

**3**

      <div>Count - {count}</div>

      <button onClick={() => dispatch("increment")}>Increment</button>

      <button onClick={() => dispatch("decrement")}>Decrement</button>

      <button onClick={() => dispatch("reset")}>Reset</button>

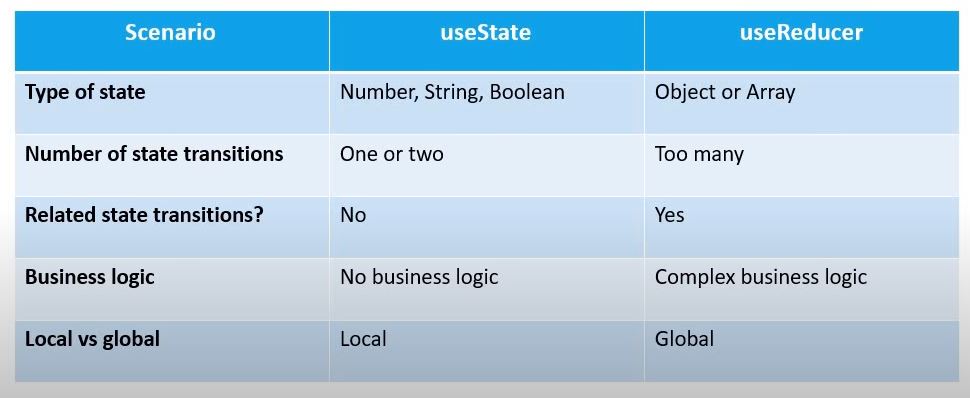
    </div>

  );

}

export default CounterOne;

1. Import useReducer from react
2. Within component call useReducer passing in – reducer method and initialState
3. Write the JSX to show the counter value with increment, decrement and reset button.
4. initialState set to zero which is the count value.
5. Now write the reducer method which accepts two parameters- current state and an action. This method returns the new state depending on the action.
6. Next, back to useReducer which returns pair of values- first the current value of the state which is the count value in this example and secondly, a dispatch method which is capable of accepting an action to execute the code specified in the render function. This is array destructuring of react.
7. We use this dispatch method to dispatch the appropriate action based on respective button click. These actions trigger the state transitions. When the state changes the component re renders and collects the counter value and displayed in the browser.

useSate vs useReducer 

Both of these hooks are used for state management in React. But there are some scenarios from where we can understand when to use useState and when to use useReducer.

1. Type of State: for the management of the primitive types like number, string, Boolean useState is the better option. Where as to manage object or array useReducer is the better choice. For example, if you are managing a count value as the state variable useState hook is your option. But if you are managing person’s data which contains first name, last name, age etc.
2. Number of state transition: if you are updating one or two variable useState is fine. But if you are updating too many variables (5 or more) then useReducer is the best choice. The main reason is it makes your state transition predictable. Using useReducer you can update several state variables in one place. It’s also become easy to maintain the code.
3. Related state transition: useReducer is better suited if the state transitions are all related or have any specific action. It also helps to write code more predictable and maintainable. If we have to manage everything with useState in different place that will be become harder to reason about.
4. Business logic: if your business logic contains complex data transition and manipulation, you better opt. useReducer. In that way all your is localized to reducer function you will have better separation of concerns. All your component has to is dispatch the appropriate action. This again make the code more readable and maintainable.
5. Local vs. global: if you want to maintain local state useState is a better choice. However if you want to maintain global state that can be altered by components deep in the component tree useReducer is a better option. Of course you will use the context hook but using the reducer hook over the state hook has a very simple advantage. With useReducer we simply have to pass one dispatch method down the component tree. That one dispatch method can update several state variables based on the action type. Where in useState we have to pass down multiple update functions, one for each state. So, for the global state management useReducer is the better choice.

**useCallback**

useCallback is a hook that will return a memorized version of the callback function that only changes if one of the dependencies has changed.

Why?

It is useful when passing callbacks to optimized child components that rely on reference equality to prevent unnecessary renders.