# React

1. React is a JavaScript library for building user interfaces.
2. React is used to build single-page applications.
3. React allows us to create reusable UI components.
4. React is a component base architecture.

Code:

import React from 'react';

import ReactDOM from 'react-dom/client';

function Hello(props) {

return <h1> Hello World! </h1>;

}

const root = ReactDOM.createRoot (document.getElementById ("root"));

root.render (<Hello />);

**Install React Project:** npx create-react-app project\_name

**React Features:**

1. JSX
2. Components
3. One-way Data Binding
4. Virtual DOM
5. Simplicity
6. Performance

**React Directly in HTML:**

<!DOCTYPE html>

<html>

<head>

<script src="https://unpkg.com/react@18/umd/react.development.js" ></script>

<script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js" ></script>

<script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>

</head>

<body>

<div id="mydiv"> </div>

<script type="text/babel">

function Hello() {

return <h1> Hello World! </h1>;

}

ReactDOM.render(<Hello />, document.getElementById('mydiv'))

</script>

</body>

</html>

# DOM in React

when html run in browser, browser parse the html file and make a DOM tree. Also parse CSS file and make CSSOM. Then DOM and CSSOM tree mixed and make render tree which print the display in browser.

**How does React Work?**

React creates a VIRTUAL DOM in memory. Instead of manipulating the browser's DOM directly, react creates a virtual DOM in memory, where it does all the necessary manipulating, before making the changes in the browser DOM. React finds out what changes have been made, and changes **only** what needs to be changed.

# Introducing JSX

1. JSX stands for JavaScript XML.
2. JSX allows us to write HTML in React.
3. JSX makes it easier to write and add HTML in React.
4. With JSX you can write JavaScript expressions inside HTML with curly braces {}.

const myElement = <h1> React is {5 + 5} </h1>;

1. inserting a Large Block of HTML

const myElement = (

<ul>

<li>Apples</li>

<li>Bananas</li>s

</ul>

);

1. Use ternary expressions instead:

const myElement = <h1> { x < 10 ? "Hello”: "Goodbye” }</h1>;

1. Since JSX is closer to JavaScript than to HTML, React DOM uses camelCase property naming convention instead of HTML attribute names.
   1. class <=> className in JSX,
   2. tabindex <=> tabIndex.

# React Render

1. React's goal is in many ways to render HTML in a web page.
2. React renders HTML to the web page by using a function called **ReactDOM.render().**

<body>

<div id="root"></div>

</body>

ReactDOM.render( <p>Hello</p> , document.getElementById('root') );

1. JSX function are return with react Render.

Const name = () =>{

<ul>

<li>Apples</li>

<li>Bananas</li>s

<li>Cherries</li>

</ul>

}

ReactDOM.render( name() , document.getElementById('root') );

1. We can also return JSX function with markup syntax:

ReactDOM.render(< name />, document.getElementById('root') );

# Components

In the past, developers had to write thousands of lines of code to build a single-page application (SPA). These applications relied on the traditional DOM structure, which made maintenance difficult—any errors required manually searching through the entire codebase to fix them.

To solve this problem, the component-based approach was introduced. In this approach, the application is divided into small, reusable, and logical units of code called components. Components are similar to functions that return HTML elements.

Components are of two types,

* + - * 1. Class components
        2. Function components

# Class Components

1. A component defined using a class is called a class component.
2. A class component must extend React.Component, which creates an inheritance chain and gives the component access to all of React’s built-in methods.
3. A class component must implement a render() method, which returns JSX.
4. Class components are also called stateful components because they can hold and manage local state.

Example:

class MyComponent extends React.Component {

render() {

return (

<div> This is main component. </div>

);

}

}

1. **State in Class Components:**

* React class components have a built-in state object.
* The state object is used to store property values that belong to the component.
* When the state changes, the component automatically re-renders to reflect the updated values.

1. **Declaring State:** React Class components have a built-in state object. The state object is where you store property values that belongs to the component.

class Car extends React.Component {

**state** = {

brand: "Ford",

model : 123,

};

render () (

return (

<h1> My Car {this.state.brand} </h1

);

}

}

1. **Using State:** Access the state anywhere in the component using **this.state.value**

class Car extends React.Component {

state = {

brand: "Ford",

model: "Mustang",

};

render () {

return (

<h1> **{ this.state.brand }** </h1>

<p> **{ this.state.model }** </p>

);

}

}

1. **Updating State:**

* To change a value in the state object, use the this.setState() method.
* When a value in the state object changes, the component will re-render, meaning that the output will change according to the new value(s).

Syntax:

**setState (updater, callback)**

* updater: Object or function that updates the state
* callback (optional): Function that runs after the update is complete

Code:

class Car extends React.Component {

state = {

brand: "Ford",

model: "Mustang",

color: "red",

};

changeColor = () => {

**this.setState(**

**{ color: "blue" } ,**

**() => {**

**console.log ("Updated count:", this.state.count);**

**}**

**);**

}

render() {

return (

<div>

<h1> My { this.state.color } </h1>

<button onClick = { this.changeColor } > Change color </button>

</div>

);

}

}

* When you want to update state based on the previous state, you should pass a function to this.setState() instead of an object.

this.setState( (state, props) => {

return {counter: state.counter + props.step};

});

1. state: this takes previous state

2. props: previous props

1. **Props in class:**

* Props (short for Properties) are used to pass data from a parent component to a child component.
* They are read-only, meaning a component cannot modify its own props.
* Props are similar to function arguments.
* In class components, props are accessed using this.props.value.

Code:

class App extends React.Component {

    render() {

      return (

            <h1> Welcome to { **this.props.name**} </h1>

      );

    }

}

export default App;

const main = <App **name=”shuvo”** />

1. **Class Constructor:**

* The constructor is a special method in a class component that runs when the component is created.
* Its main purposes are:

1. Initialize the component’s state
2. Bind event handlers (if not using arrow functions)
3. Access props using this.props

* The constructor is called before the component mounts.
* When using a constructor in a React class component, you must call super(props) at the beginning.
  1. Without super(props), this.props will be undefined inside the constructor, leading to errors.

Syntax:

Constructor(props) {

**super(props);**

}

1. **Constructor and state:**

* You cannot call setState() method directly in the **constructor().**
* If the component needs to use local state, you need directly to use **'this.state'** to assign the initial state in the constructor.
* The constructor only uses this.state to assign initial state, and all other methods need to use set.state() method.

Code:

class App extends Component {

**constructor(props){**

**super(props**);

**this.state** = {    data: 'www.javatpoint.com'    }

    this.handleEvent = this.handleEvent.bind(this);

  }

  handle(){

    console.log( this.props );

  }

  render() {

     return (

       <div className="App">

     <input type ="text" value={this.state.data} />

         <button onClick = { **this.handle** }> Please Click </button>

       </div>

     );

  }

}

export default App;

1. **Children In Class:**

* Children is a special prop used to pass content from a parent component to its child component.
* The content must be placed between the parent component’s opening and closing tags.
* It allows you to wrap or nest elements inside a component.

Syntax:

Const Name = ( props )=>{

Return ( { **props.children** } );

}

<Name>  **value** </Name>

## React Component Life-Cycle:

Each component has several “lifecycle methods” that you can override to run code at times in the process.

1. Mounting Phase
2. Updating Phase
3. Unmounting Phase

### **Mounting Phase**

**When an instance of a component is being created and inserted into DOM is called mounting methods.**

1. **constructor ():**

* Initializes the component.
* Sets up state and binds methods.
* Called before mounting.

**Syntax**: constructor(props)

**Code:**

constructor(props) {

super(props);

this.state = { counter: 0 };

this.handleClick = this.handleClick.bind(this);

}

1. **componentDidMount():**

* Called after the component is mounted.
* Ideal for:

1. Fetching data from APIs
2. Setting up subscriptions
3. Interacting with the DOM

Code:

class App extends React.Component {

    constructor(props) {

      super(props);

      this.state = { name: "shuvo"};

    }

**componentDidMount()** {

console.log('Component Did MOUNT!')

    }

    render() {

return ( <h1> { this.state.name} </h1> );

}

}

1. **static getDerivedStateFromProps(props, state):**
2. **render ():**
3. The render method will be called each time an update happens
4. If you don't want to render anything, you can return a null or false value.

class App extends React.Component {

**render () {**

 return (

<h1> I am render method </h1>

);

}

}

### **Updating Phase**

When a component is being re-rendered as a result of changes to either its props or state

1. **static getDerivedStateFromProps(props, state):**
2. **shouldComponentUpdate (nextProps, nextState)**

* It is invoked when a component decides any changes/updating to the DOM.
* Determines if the component should re-render.
* Return true or false.

1. **getSnapshotBeforeUpdate ():**
2. **componentDidUpdate (prevProps, prevState, snapshot)**

* componentDidUpdate () is invoked immediately after updating occurs.
* This method is not called for the initial render.

**Syntex**:

componentDidUpdate( prevProps, prevState, snapshot)

**preprops**: it takes previous props of a component

**prestate** : take previous state of component .

**class** **App** **extends React.Component** {

**constructor**(props) {

      super(props);

      this.state = { name: "shuvo"};

   }

**componentDidUpdate**(prevProps, prevState) {

**console**.log('Component Did UPDATE!')

   }

**render**() {

**return** (

<h1> { this.state.name} </h1>

);

}

}

1. render ()

* The render() method is invoked to examine this.props and this.state and determine what should be displayed on the UI.
* It must return one of the following:

1. React elements
2. Arrays or fragments
3. Booleans or null (renders nothing)
4. Strings or Numbers

* If shouldComponentUpdate() returns false, the render() method still runs internally to ensure the component’s UI stays consistent.

### Unmounting Phase

When a component is being removed from the DOM

1. **componentWillUnmount():**
2. It is invoked immediately before a component is unmounted and destroyed.
3. Perform cleanup:

* Cancel timers
* Remove subscriptions
* Abort network requests

class App extends React.Component {

   constructor(props) {

      super(props);

      this.state = {name: "shuvo"};

   }

   componentWillUnmount() {

      console.log('Component Will UNMOUNT!')

   }

   render() {

      return (

         <div>

             <h3>Hello {this.state.hello}</h3>

             <button onClick = {this.changeState}>Click Here!</button>

         </div>

      );

   }

}

export default App;

### Error Handling

When there is an error during rendering, in a lifecycle method, or in the constructor of any child component

1. **static getDerivedStateFromError():**
2. **componentDidCatech:**

# Functional Component

1. Functional components are a way to create components that only contain a render method and do not have their own state (before Hooks).
2. They are simple JavaScript functions that may or may not receive data as parameters.
3. Functional components are also called stateless components because they do not manage local state by default.

**function WelcomeMessage() {**

**return <h1>Welcome to the world </h1>;**

**}**

**Props in Functional Components:**

Props can be passed to functional components just like class components, but here they are received as function arguments.

function Car(**props**) {

return <h2> I am a { props.brand } </h2>;

}

const myElement = <Car brand="Ford" />;

# Events handle

An event is an action that could be triggered as a result of the user action or system generated event. For example, a mouse clicks, loading of a web page, pressing a key, window resizes, and other interactions are called events.

React has its own event handling system which is very similar to handling events on DOM elements. The react event handling system is known as Synthetic Events. The synthetic event is a cross-browser wrapper of the browser's native even

**Rules:**

1. React events are named as camel Case instead of lowercase.
2. With JSX, a function is passed as the event handler instead of a string.
3. React event handlers are written inside curly bracket { }:

function Football() {

**const shoot = ()** => {

alert("Great Shot!")

}

return (

<button onClick**={shoot}>**Take the shot!</button>

);

}

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render( <Football /> );

1. To prevent default behavior, call event.preventDefault().

function ActionLink() {

     function handleClick(e) {

**e.preventDefault();**

        console.log('You had clicked a Link.');

     }

     return (

        <a href="#" onClick={ handleClick }> Click\_Me    </a>

     );

}

1. To pass an **argument** to an event handler, use an arrow function.

function Football() {

const shoot = (a) => {

alert(a);

}

return (

<button onClick={ () => **shoot("Goal!") }**> shot </button>

);

}

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Football />);

1. Event handlers have access to the React **event** that triggered the function.

function Football() {

const shoot = ( a, **b )** => {

alert( b.**type** );

}

return (

<button onClick={ (**event**) => shoot("Goal!", **event**) } > Take the shot </button>

);

}

## This problem in event handle function:

* In class components, event handling often faces the problem of this binding.
* when we want to set a value in setState() method in event function, react face a problem for reference value of this.

class App extends React.Component {

     constructor(props) {

        super(props);

        this.state = {

             count : 1

        };

     }

     handle(event) {

**this.setState**( { **count** : count +1 });

     }

      render () {

         return (

       <button type="text"  onClick ={ this**.handle** } >   click </button>

         );

     }

}

export default App;

## The way to solve problem:

1. **arrow function:** If we convert the function in **arrow** **function**, then reference value this not create a problem.

class App extends React.Component {

    constructor(props) {

        super(props);

        this.state = {

             count : 1

        };

    }

**handle** = () => {

**this.setState**({

**count** : count +1

        });

    }

    render () {

        return (

                <button type="text"  onClick ={ this**.handle** }>   click </button>

        );

    }

}

export default App;

1. **Bind method:** Bind the function in construction function. Then this not create problem.

class App extends React.Component {

    constructor(props) {

        super(props);

        this.state = {

             count : 1,

this.handle = this.handle.bind(this)

        };

    }

    handle = {

        this.setState({

            count : count +1

        });

    }

    render () {

        return (

                <button type="text"  onClick ={ this.handle } >  click <button>

        );

    }

}

export default App;

1. **bind in inline event:** we can use bind method in inline html. In inline bind method we can also pass parameter in function.

**Syntax:** this.fucntionName.bind( this , param)

Code:

 handle = (param) {

        this.setState({

console.log(param)

        });

    }

            <button type="text” onClick = { this.handle.bind( this , param) } >  click </button>

# Conditional Rendering

In React, we can create multiple components which encapsulate behavior that we need. After that, we can render them depending on some conditions or the state of our application. In other words, based on one or several conditions, a component decides which elements it will return.

There is more than one way to do conditional rendering in React. They are given below.

* + - 1. if
      2. ternary operator
      3. logical && operator
      4. switch case operator

1. **if Statement:** We can use the if JavaScript operator to decide which component to render.

Code:

function Goal(props) {

const isGoal = props.isGoal;

if (isGoal) {

return <MadeGoal />;

}

return <MissedGoal />;

}

1. **Logical && operator:** This operator is used for checking the condition. If the condition is true, it will return the element right after &&, and if it is false, React will ignore and skip it. function Garage(props) {

Code:

const cars = props.cars;

return (

<>

<h1>Garage</h1>

{ cars.length > 0 && <h2> shuvo </h2> }

</>

);

}

const cars = ['Ford', 'BMW', 'Audi'];

<Garage cars={cars} />

1. **Ternary Operator:** The ternary operator is used in cases where two blocks alternate given a certain condition. This operator makes your if-else statement more concise. It takes three operands and used as a shortcut for the if statement.

Syntax: condition ?  true : false

code:

function Goal(props) {

const isGoal = props.isGoal ;

return { isGoal ? <MadeGoal/> : <MissedGoal /> }

}

root.render( <Goal isGoal= { false } /> );

1. **Switch case operator:**

Code:

function NotificationMsg({ text}) {

   switch(text) {

    case 'Hi All':

      return <Message: text={text} />;

    case 'Hello JavaTpoint':

      return <Message text={text} />;

    default:

      return null;

   }

}

1. **preventing Component from Rendering:** If we want to control component rendering (return or not) inside from component then we return null or component.

function Show(props) {

If ( !props.displayMessage )

return null;

else

return <h3>Component is rendered</h3>;

}

<Show displayMessage = {true}  />

# Lists

1. Lists are used to display data in an ordered format. Example - display menus on websites.
2. In React, Lists can be created in a similar way as we create lists in JavaScript.

var numbers = [1, 2, 3, 4, 5];

1. map() function is used for traversing the lists. here map take every element of array and called the function every time.

const myList = ['Peter', 'Sachin', 'Kevin', 'Dhoni', 'Alisa'];

const listItems = myList.map( (myList) => {

    return <li> { myList } </li>;

});

 <ul> { listItems } </ul>,

**React Keys:**

1. A key is a unique identifier.
2. In React, it is used to identify which items have changed, updated, or deleted from the Lists.
3. It is useful when we dynamically created components or when the users alter the lists.
4. Keys should be given inside the array to give the elements a stable identity.
5. Array index should not use as a key.

const stringLists = [ 'Peter', 'Sachin', 'Kevin', 'Dhoni', 'Alisa' ];

const updatedLists = stringLists.map( (strList) => {

    <li  key = { strList.id } > { strList } </li>;

});

function Car(props) {

return <li> I am a { props.brand } </li>;

}

function Garage() {

const cars = [

{id: 1, brand: 'Ford'},

{id: 2, brand: 'BMW'},

{id: 3, brand: 'Audi'}

];

return (

<ul> { cars.map( **(car) => <Car key={car.id} brand = {car.brand}** /> ) } </ul>

);

}

< Garage />

# style

1. CSS Stylesheets:

**MyStyle.css =>**

. primary {

    color:orange

}

**Person.jsx =>**

import "../component/MyStyle.css";

function Person () {

  return <h1 className="primary"> ddd </h1> ;

}

export default Person;

1. Inline Styling:

const heading = {

fontSize : ‘72px’,

color : ‘red’

}

function Person () {

  return <h1 style={heading}> ddd </h1> ;

}

export default Person;

1. CSS Modules:

**myStyle.module.css =>**

. primary {

    color:orange

}

**Person.jsx =>**

Import style "../component/myStyle.module.css";

function Person () {

  return <h1 className={style.primary}> ddd </h1> ;

}

export default Person;

\*\*\* Global CSS styles can be used in child components, but CSS Modules cannot be accessed directly by child components. As a result, styles defined in a parent component's module are not applied to child components, and parent styles may be overridden.

1. CSS in JS Libaries:

# Form

Browser has a state. This state maintains the form element (<input>, <textarea> and <select> etc) insert, update on base of user input. But react also have a state in component for data handling. when we work in react, browser and react both have state and want to work their state which make conflict. So that there are two way maintain the form element.

1. Controlled component
2. Uncontrolled component

# Controlled Component

When form element whose value is controlled by React in this way is called a “controlled component”. When the data is handled by the react components, all the data is stored in the component state. We handle the data in react by setState() method.

1. **Value:**  When we use value attribute in form element, react know that this is controlled component. It controlled by react.

<input type="text" **value={ this.state.value }** />

1. **Read-only mood :** In react, all DOM event handle by it. So, if we do not do any event, it takes form element as read only mode.

<input type="text" **value=”shuvo”** />

1. **State change:** In react component state are change by setState method. if we want to change form element value, we want to change the state value.

State = {

name: “shuvo”

}

handle = (e) => {

this.setState({

     name: e.target.value,

});

}

<input type="text" **value={ this.state.name }** onChange={this.handle} />

**Form in react:**

1. In React, a <textarea> uses a value attribute instead. This way, a form using a <textarea> can be written very similarly to a form that uses a single-line input:
2. In React, the selected value is defined with a value attribute on the select tag

export default class From extends Component {

  state = {

    input: "hello",

    text: "text",

    select: "",

    checkbox: true,

  };

  handle = (e) => {

if (e.target.type === "text") {

this.setState({

        input: e.target.value,

});

} else if (e.target.type === "textarea") {

 this.setState({

        text: e.target.value,

 });

} else if (e.target.type === "select-one") {

 this.setState({

        select: e.target.value,

 });

} else if (e.target.type === "checkbox") {

this.setState({

        checkbox: e.target.value,

 });

} else {

console.log("lfk");

    }

  };

  render() {

    const { input, text, select, checkbox } = this.state;

    return (

        <form>

            <input type="text “ value={input}  onChange={this.handle}   />

            <h1> TEXT AREA </h1>

            <textarea name="text" value={text} onChange={this.handle} />

            <h1> Select BOX </h1>

            <select value={select} onChange={this.handle}>

              <option value="1">name</option>

              <option value="2">roll</option>

            </select>

            <h1> check BOX </h1>

            <input type="checkbox" value={checkbox} onChange={this.handle} />

        </form>

    );

  }

}

**Submitting Forms:**

You can control the submit action by adding an event handler in the onSubmit Attribute for the <form>.

Code:

function MyForm() {

const [name, setName] = useState(" ");

const handleSubmit = (event) => {

event.preventDefault();

alert(`The name you entered was: ${name}`)

}

return (

<form **onSubmit={handleSubmit}>**

<label>Enter your name: </label>

<input

type="text"

value={name}

onChange={(e) => setName(e.target.value)}

/>

<input type="submit" />

</form>

)

}

**The file input Tag:**

Input file tag should be controlled by uncontrolled component. because in file there are no value attribute.

<input type="file" />

**Handling Multiple Inputs:**

1. When you need to handle multiple controlled input elements, you can add a name attribute to each element and let the handler function choose what to do based on the value of event.target.name.
2. Without <select> and <checkbox> other form element handles by **event.target.name** and **event.target.value**

export default class From extends Component {

  state = {

    input: "hello",

    text: "text",

  };

  handle = (e) => {

    setState( { e.target.name : e.target.value } )

  };

  render() {

    const { input, text, select, checkbox } = this.state;

    return (

        <form>

            <input type="text “ value={input}  onChange={this.handle}   />

            <textarea name="text" value={text} onChange={this.handle} />

        </form>

    );

  }

}

**Input Null Value:** If we want to editable form but not want value, then we set null or undefined.

<input type=”text” value ={null} />

# Uncontrolled component

1. The uncontrolled input is like the traditional HTML form inputs. DOM itself handles the form data.
2. The HTML elements maintain their own state that will be updated when the input value changes.
3. To write an uncontrolled component, you need to use a ref to get form values from the DOM.

Code:

class App extends React.Component {

constructor(props) {

      super(props);

      this. handle = this. handle.bind(this);

      this.input = React.createRef();

}

 handle (event) {

      alert('You have entered  successfully.');

      event.preventDefault();

}

render() {

 return (

      <form onSubmit={this.handle}>

        <input name=”name” type="text" ref={this.input} />

        <input name=”date” type="text" ref={this.input} />

        <input type="submit" value="Submit" />

      </form>

);

 }

}

**Ref:**

1. Refs are a function provided by React to access the DOM element and the React element that you might have created on your own.
2. They are used in cases where we want to change the value of a child component, without making use of props and all.
3. They also provide us with good functionality as we can use callbacks with them.
4. Refs are created using React.createRef() and attached to React elements via the ref attribute.

const refContainer = React.createRef();

1. When a ref is passed to an element in render, a reference to the node becomes accessible at the current attribute of the ref.

const node = this.myRef.current

1. You may not use the ref attribute on function components because they don’t have instances.

**Code:**

class CustomTextInput extends React.Component {

constructor(props) {

super(props);

this.textInput = React.createRef();

this.focusTextInput = this.focusTextInput.bind(this);

}

handle() {

this.textInput.current.focus();

}

render() {

return (

<input type="text" ref={this.textInput} />

<input type="button" value="Focus" onClick={this.handle} />

);

}

}

# Lifting state:

In react we make component for same work in multiple time in various reference. When we need to send state for one component to another component. But if component is in parallel then we cannot send state as props in component.



Here component A export component B, C. here component B take input but cannot send to component C. So, it is a problem to send data to other component. to solve this, we use lifting state up.

In lifting state we send all the state in parent component. Then we send the state as props in child component. It is called lifting state

A

-State

-method

B

-props.state

-props.method

C

-props.state

-props.method

Here all state and function are in top level component A. then with props component sent to B and C. it is called lifting state up.

**Code:**

**File B:**

import React,{ Component } from 'react';

class B extends Component {

constructor(props) {

    super(props);

    this.handleText = this.handleText.bind(this);

}

handleText (e){

     this.props.handleTextChange(e.target.value);

}

render() {

return  <input value={this.props.text}   onChange={this.handleText} />;

}

}

export default B;

**File C:**

import React,{ Component } from 'react';

class C extends Component {

render() {

     return  <h3>Output: {this.props.text}</h3> ;

}

}

export default C;

**Parent:**

class A extends Component {

   constructor(props) {

    super(props);

    this.handleText= this.handleText.bind(this);

    this.state = {text: ''};

   }

   handleText (newText) {

     this.setState({text: newText});

   }

   render() {

    return (

          <B text={this.state.text}  handleTextChange={this.handleText} />

          <C text={this.state.text} />

    );

   }

}

# Composition vs Inheritance

**Inheritance:**

Inheritance is a mechanism in which one object acquires all the properties and behaviors of a parent object.

Code:

class Car {

  constructor(brand) {

    this.carname = brand;

   }

  present() {

    return 'I have a ' + this.carname;

   }

}

class Model extends Car {

   constructor (brand, mod) {

    super(brand);

    this.model = mod;

   }

  show () {

    return this.present() + ', it is a ' + this.model;

   }

}

**Problem of inheritance:**

1. In inheritance, react child component extends all parent component method. If child want to use single method of parent component it not possible.
2. component is tightly coupled.
3. from child it not clear what parents does.

**Composition:**

The functionality is quite similar to inheritance, however rather than inheriting the properties of the parent class it represents a class that references one or more objects of the other class as an instance.  This way it provides the new behavior to the class or object rather than extending it in the hierarchy.

In React, we can make components more generic by accepting props, which are to React components what parameters are to functions.

Component composition is the name for passing components as props to other components, thus creating new components with other components.

Code:

const Button = ({ onClick, children }) => (

<button onClick={onClick}>{children}</button>

);

const App = () => {

const onClick = () => alert ('Hey 👋');

return (

<Button onClick={onClick}>Click me! </Button>

);

};

# Pattern of share State of react

To share same functionality in all component there are two patterns.

1. higher order component
2. props rendering

# Higher-Order Components

* higher-order component is a function that takes a component and returns a new component. Where as a component transforms props into UI, a higher-order component transforms a component into another component.

const EnhancedComponent = higherOrderComponent(WrappedComponent);

* when we use similar function in various component, we use higher-order component. we can solve it by lifting state up. In lifting state up, we send component in props. If want to send very deep component it is very bad. So, in react some time lifting state up is not good for component. so we use HOC in react.
* If we want to more add component/event in component it is very difficult to add component in every other component. With higher order component is very easy.

function add (a, b) {

  return a + b

}

function higherOrder(a, addReference) {

  return addReference(a, 20)

}

//Function call

higherOrder(30, add) // 50

**Component without HOC:**

A

B

D

C

D

RETURN

**Component with HOC:**

A

D

RETURN

B

C

**Code:**

**Hoc function:**

const WithCounter = (OriginalCom) => {

  class NewCom extends React.Component {

    state = {

      count: 0,

    };

    handle = () => {

      this.setState((m) => ({

        count: m.count + 1,

      }));

    };

render() {

        const {count} = this.state;

        return <OriginalCom count={count} handle={this.handle} />

    }

  }

  return NewCom;

};

export default WithCounter;

**other function:**

import WithCounter from "./WithCounter";

const ClickCounter = (props) => {

    const {count , handle} = props;

    return (

<div>

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

    click {count} times

    </button>

</div>

      );

};

export default WithCounter(ClickCounter);

**main function:**

function App() {

return <ClickCounter />

}

We create new component with a state, that use in various component. Then we pass all the component in new state component and become other component.

# Props

1. The Render Props is a technique in ReactJS for sharing code between React components using a prop whose value is a function.
2. Render props is a prop which decide logic in props.
3. Child component takes render props as a function and calls it instead of implementing its own render logic.
4. In render props, we pass props in child component that we want to render.

Code:

Const ChildrenComponent= (render) => {

return   render;

}

Const ParentComponent = () => {

Return <ChildrenComponent render = {I am render Props} />

}

1. We can decide in props which component should return from children component.

Const ChildrenComponent= ( render ) => {

return   render;

}

Const ParentComponent =()=>{

Const name =true;

Return(

< ChildrenComponent

render ={ (name)=> ( name ? “i am “ : “I am not “ ) }

/>

)

}

# Context

1. Context allows passing data through the component tree without passing props down manually at every level.
2. Context is primarily used when some data needs to be accessible by many components at different nesting levels
3. In React application, we passed data in a top-down approach via props. Sometimes it is inconvenient for certain types of props that are required by many components in the React application. Context provides a way to pass values between components without explicitly passing a prop through every level of the component tree.

**React Context API:**

1. React.createContext
2. Context.provider
3. Context.Consumer
4. Class.contextType
5. **React.createContext**: we are creating context with React.createContext()

Const Context = React.createContext( defaultValue );

1. **Context.Provider:** Wrap child components in the Context Provider and supply the state value.

<Context.Provider value= {/\* some value \*/}>

<Component user={user} />

</ Context.Provider >

1. **Context.Consumer:**
2. wherever we want to consume (or use) what was provided on our context, we use the consumer component.
3. Consumer component take a function only

< Context.Consumer >

       { value =>  context value }

</ Context.Consumer >

1. **Class.contextType:** The contextType property on a class used to assign a Context object which is created by React.createContext(). It allows you to consume the closest current value of that Context type using this.context. We can reference this in any of the component life-cycle methods, including the render function.

**code:**

import React from 'react';

export const UserContext = React.createContext();

export default function App() {

return (

<UserContext.Provider value="Reed">

<User />

</UserContext.Provider>

)

}

function User() {

return (

<UserContext.Consumer>

{value => <h1>{value}</h1>}

</UserContext.Consumer>

)

}

# Memo

1. memo use in functional component.

2. memo use for skip rendering a component if its props have not changed.

Code:

import { memo } from "react";

const Todos = ({todos}) => {

console.log ("child render");

return (

<>

<h2>My Todos</h2>

{ todos.map( (todo, index) => {

return <p key={index}> {todo} </p>;

}) }

</>

);

};

export default memo (Todos);

# ForwardRef

1. Ref forwarding is a technique for automatically passing a ref through a component to one of its children.
2. This is typically not necessary for most components in the application.
3. However, it can be useful for some kinds of components, especially in reusable component libraries.

FancyBUtton.js:

const FancyButton = (props, ref) => (

<button ref={ref} className="FancyButton"> {props.children} </button>

));

Export default React.forwardRef(FancyButton);

Main.js

const ref = React.createRef();

<FancyButton ref={ref} > Click me! </ FancyButton>;

# HOOK

1. Hooks allow function components to have access to state and other React features
2. hook use in top level of code.
3. hook use in react function only.
4. Do not call hook inside loop, conditions, nested function

**Basic Hooks**

1. useState
2. useEffect
3. useContext

**Additional Hooks**

1. useReducer
2. useCallback
3. useMemo
4. useRef
5. useImperativeHandle
6. useLayoutEffect
7. useDebugValue

# useState()

1. The React useState() function is a Hook that allows us to track state in a function component.
2. useState() generally refers to data or properties that need to be tracking in an application.

**Syntax: const [state, stateFucntion] = useState();**

1. **Defined:** As a normal function, we can define useState() as below for understanding-----

Function:

Const useState = (argument) => {

Value = argument

………………………………………

return [ value, function];

}

useState(0);

1. **Set Initial Value:**
2. useState () hook take only one argument. It takes initial value as an argument.
3. In useState we send number, string, object, array, function as initial state.

const [age, setAge] = useState (42);

const [fruit, setFruit] = useState('banana');

const [todos, setTodos] = useState ([ {text: 'Learn Hooks’} ] );

1. **Return value:**
2. UseState method return an array.
3. Array first index are state value;
4. Array second index are function. Which change the state.

const array = useState(initialValue)

state = array [0]

setState = array [1].

1. We can use array destructure for argument.

const [ argument\_state , argument\_function] = useState (argument);

1. **State change:**
2. we update the state value by second return value called argument\_function. this is similar to this.setState in a class.

Syntax: const [ argument\_state, argument\_function] = useState (argument);

Code:

function FavoriteColor() {

const [color, setColor] = useState("red");

const handle = () => {

setCount( color=”black” );

}

return <h1> My favorite color is {color}! </h1>

}

1. If we want to change the value with help of previous state, we pass the function as argument. where function takes previous state as an argument.

Syntax:

( preValue / preState ) => { preValue + 1 }

argument\_function( ( preValue ) => { preValue +1 } );

Code:

import React, {useState} from 'react';

function Example () {

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

const handle = () => {

setCount( (m)=> ( m+1 ) )

}

return (

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

<button onClick={handle}> Click me </button>

);

}

1. Calling the set function does not change the current state in the already executing code: It only affects what useState will return starting from the next render.

function handleClick() {

setName('Robin');

console.log(name); // Still "Taylor"!

}

#### Updating Objects and Arrays in State

**Problem:** You can put objects and arrays into state. In React, state is considered read-only, so you should replace it rather than mutate your existing objects. If we update the object or array with argument\_function, then it changes all the object and array.

Code:

function Car () {

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

return (

<div>

<p> { name.firstName } { name.lastName } </p>

<input type=”text”

value={name.firstName }

onchange = {e => setName ({ firstName: e.target.value})}

/>

<input type=”text”

value={name.lastName}

onchange={ e => setName ({ lastName: e.target.value})}

/>

</div>

)

}

Here when we use first name input then all object of name is change by first name input.

**Solve :** To solve this, we use spread operator (...). spread syntax ensures that the state object is replaced rather than mutated.

Code:

function Car() {

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

return (

<div>

<p> { name.firstName } {name.lastName } </p>

<input type=”text”

value={name.firstName}

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

/>

<input type=”text”

value={name.lastName}

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

/>

</div>

)

}

# useEffect()

**React are works for –**

1. render UI

2. React on user input and action

3. Render JSX

4. manage state & props

5. Evaluate State and props change

**React also do various work (side effect)**

1. fetching data from API

2. updating Dom

3. setting any Subscription or timer

**The side effect handle in class component with the method of-**

1. componentDidMount ()

2.componentDidUpdate ()

3.componentWillUpdate ()

**Here is some problem to use this method –**

1. repeating code

2. unorganized code

In useEffect we solve all problem in functional component.

**useEffect():**

1. Use Effect is a function which run every render.
2. Help us perform side Effect in functional components
3. Solves all the problem of lifecycle methods in class component
4. We do not repeat the code.
5. We use useEffect Hook, to do something after render in component.
6. useEffect use inside the component for access the any props or variable.

**Syntax:**

useEffect (peram\_1, peram\_2(**dependency**));

**param\_1:** take a function.

**Param\_2:** it is optional. It is a dependency. It is use for when is useEffect called or not.

 Dependencies argument:

* Run Every Render: If we run useEffect hook in every render we do not use Dependency.

function MyComponent() {

const [data, setData] = useState([])

useEffect ( () => {

fetchData().then( myData => setData(myData))

} );

}

* If param\_2 == [ ] then useEffect called only one time. It is as like componentDidMount.

useEffect(() => {

document.title = `You clicked ${count} times`;

// Runs ONCE after initial rendering

}, [] );

* Ifparam\_2 == [props or state], then it called when props or state is change. It is as like componentDidUpdate.

Example:

function MyComponent( { prop } ) {

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

useEffect(() => {

// Runs ONCE after initial rendering

// and after every rendering ONLY IF `prop` or `state` changes

}, [prop, state]);

}

Code:

function Counter() {

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

useEffect(() => {

setCount(() => count \* 2);

}, [count]);

return (

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

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

);

}

For Cleanup:

Some effects require cleanup to reduce memory leaks.  For example, if you have a countdown timer using the setInterval function, that interval will not stop unless we use the clearInterval function. We use componentWillUnmount for cleanup in class component. In Hook, for cleanup the effect we return the function.

**useEffect (() => {**

**// Side-effect...**

**return function cleanup() {**

**// Side-effect cleanup...**

**};**

**}, dependencies);**

useEffect() in practice:

1. Fetching data: Can perform data fetching side-effect.

import { useEffect, useState } from 'react';

function FetchEmployees() {

const [employees, setEmployees] = useState([]);

useEffect(() => {

async function fetchEmployees() {

const response = await fetch('/employees');

const fetchedEmployees = await response.json(response);

setEmployees(fetchedEmployees);

}

fetchEmployees();

}, []);

return (

<div>

{ employees.map (name => <div>{name}</div>)}

</div>

);

}

# useContext

1. useContext are use alternative of context.consumer component.
2. We can access value by useContext() hook .

const value = useContext(MyContext);

**Example:**

**function Component1**() {

const [user, setUser] = useState("Jesse Hall");

return (

**<UserContext.Provider value={user}>**

<h1> { `Hello ${user}!` } </h1>

<Component2 user={user} />

**</UserContext.Provider>**

);

}

**function Component5()** {

**const user = useContext(UserContext);**

return (

<h2> {`Hello ${user} again! `} </h2>

);

}

**Using the context in React requires 3 simple steps**:

* + - 1. creating the context,
      2. providing the context,
      3. useContext() .
* **Creating the context:** The built-in factory function createContext(default) creates a context instance:

**import { createContext } from 'react';**

**const Context = createContext('Default Value');**

* **Providing the context:**

1. Context.Provider component available on the context instance is used to provide the context to its child components, no matter how deep they are.
2. To set the value of context use the value prop available on the <Context.Provider value={value} />:
3. All the components that'd like later to consume the context must be wrapped inside the provider component.
4. To change the context value, simply update the value prop.

function Main () {

const value = 'My Context Value';

return (

<Context.Provider value= {value}>

<MyComponent />

</Context.Provider>

);

}

* **Consuming the context**

1. We can consume the context value by use the useContext(Context) React hook:
2. The hook returns the value of the context: value = useContext(Context).
3. Component will re-render when the context value changes.

import { useContext } from 'react';

function MyComponent() {

const value = useContext(Context);

return <span> {value} </span>;

}

# useCallback

The React useCallback Hook returns a memorized callback function.

**Syntax:**

const addTodo = useCallback( param\_1, param\_2 );

param1: take a function.

useCallBack ( () => { document.title = `You clicked ${count} times` ; } );

param2:

1. dependency variable , for which useCalled called or not.

Example:

import { useState, useCallback } from "react";

import ReactDOM from "react-dom/client";

import Todos from "./Todos";

const App = () => {

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

const [todos, setTodos] = useState([]);

const increment = () => {

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

};

const addTodo = useCallback( () => {

setTodos((t) => [...t, "New Todo"]);

}, [todos] );

return (

<>

<Todos todos={todos} addTodo={addTodo} />

<hr />

<div>

Count: {count}

<button onClick={increment}>+</button>

</div>

</>

);

};

# useMemo

1. The React useMemo Hook returns a memorized value. The useMemo Hook only runs when one of its dependency’s updates.

**Syntax:**

const addTodo = useMemo( param\_1, param\_2 );

param1: take a function.

useCallBack ( () => { document.title = `You clicked ${count} times` ; } );

param2: dependency variable , for which useCallback called or not.

**Example:**

const App = () => {

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

const [todos, setTodos] = useState([]);

const calculation = useMemo( ()=> {

for (let i = 0; i < 1000000000; i++) {

num += 1;

}

return num;

}, [count]);

const increment = () => {

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

};

const addTodo = () => {

setTodos((t) => [...t, "New Todo"]);

};

return (

<div>

<div>

<h2>My Todos</h2>

{ todos.map ( (todo, index) => { return <p key={index} > {todo} </p>; }) }

<button onClick = { addTodo }> Add Todo </button>

</div>

<hr />

<div>

Count: {count}

<button onClick={increment}>+</button>

<h2>Expensive Calculation</h2>

{calculation}

</div>

</div>

);

};

# useRef

1. The useRef is a hook that allows to directly create a reference to the DOM element in the functional component.
2. The useRef returns a mutable ref object. This object has a property called **.current.**
3. The value is persisted in the **refContainer.current** property.

Syntax: const refContainer = useRef(initialValue);

code:

**function** **App**() {

**const** **focusPoint** = useRef(null);

**const** **onClickHandler** = () => {

**focusPoint**.**current**.value

**focusPoint**.current.focus();

  };

**return** (

**<button** **onClick**={onClickHandler}>

     <**textarea** ref={ focusPoint } />

  );

};

# useReducer

1. The useReducer() Hook is like the useState() Hook.
2. It allows for custom state logic.

**Syntax:** useReducer(<**reducer**>, < **initialState** >, <init>)

1. **Reducer:** The reducer function contains your custom state logic.it take two arguments.

Reducer (current **state**, **action**)

**Current** **state**: value of state.

**Action**: event of component.

1. **Initial State**: the initialState be a simple value but generally will contain an object.
2. **Init:** You can also create the initial state lazily. To do this, you can pass an init function as the third argument. The initial state will be set to init(initialArg)

**Return value of Reducer:** useReducer return two values. State and dispatch.

const [state, dispatch] = useReducer(<**reducer**>, < **initialState** >);

**code:**

**const initialState = {count: 0};**

**function reducer(state, action**) {

**switch (action.type) {**

**case 'increment':**

**return {count: state.count + 1};**

**case 'decrement':**

**return {count: state.count - 1};**

**default:**

**throw new Error();**

**}**

}

**function Counter() {**

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

**return (**

**<>**

**Count: {state.count}**

**<button onClick={() => dispatch({type: 'decrement'})}>-</button>**

**<button onClick={() => dispatch({type 'increment'})}>+</button>**

**</>**

**);**

}

# React Router (V5 )

To add React Router in your application

npm i -D react-router-dom

1. Basic Usage:

import ReactDOM from "react-dom/client";

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

import Layout from "./pages/Layout";

import Home from "./pages/Home";

export default function App() {

return (

<BrowserRouter>

<Route path="/" element={Layout} />

<Route path=”/home” element={ Home } />

</BrowserRouter>

);

}

1. Exact Route:
2. If route is “/” then it open “/”, “/home” both components . because in react route match with Browser Route.
3. to solve this we use exact in Route. With exact keyword we Route as accurate route.

<BrowserRouter>

<Route exact path="/" element={Layout} />

<Route exact path=”/home” element={ Home } />

</BrowserRouter>

1. Not found route:

<BrowserRouter>

<Route exact path="/" element={Layout} />

<Route exact path=”/home” element={ Home } />

<Route component={ Error } />

</BrowserRouter>

1. Switch: If we want to found first route, we use switch.

<BrowserRouter>

<switch>

<Route exact path="/" element={Layout} />

<Route exact path=”/home” element={ Home } />

<Route component={ Error } />

<switch>

</BrowserRouter>

1. Dynamic Route:

<Route exact path="/" element={Layout} />

<Route exact path=”/home/: id” element= {Home} />

1. Props in Route Component:

<Route exact path="/"> <Layout handle={add} /> <Route/>

<Route exact path=”/home/: id” > <Home /> <Route/>

1. Route in Render Props:

<Route exact path="/" render={ ()=> <Layout name=”shuvo” />} />

1. Redirect Router:

<Route exact path="/" >

<Redirect to=”/home” />

<Route>

1. Link

<Link to="/home ">Home</Link>

<Link to="/ ">layout</Link>

1. Link Parameter:

<Link to="/home?name=shuvo ">layout</Link>

1. Link Object:

<Link to= {{

Pathname: “/home”,

Search: “?name=shuvo”,

Hash: “mt5”,

State: { status: true }

}} />

1. NavLink: To control style, we use NavLink.

<NavLink exact

to="/home?name=shuvo"

activeStyle={{

fontWeight : ‘blod’,

color : ‘red’

}}

> layout </NavLink>

# React Router (V6)

In react router V6-

1. No **switch** statement.
2. No **Exa**ct keyword
3. No **redirect** keyword
4. Basic Usage:

import ReactDOM from "react-dom/client";

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

import Layout from “. /pages/Layout";

import Home from “. /pages/Home";

export default function App () {

return (

<BrowserRouter>

<Route path="/” element= {<Layout />}

<Route path=”/home” element= {<Home />} />

<Route path=”/about” element= {<Navigate to=”/contact” />} />

</BrowserRouter>

);

}

1. Keyword:

<BrowserRouter>

<Routes>

<Route path="/” element= {<Layout />} />

<Route path=”/home/\*” element= {<Home />} />

<Route path=”/home/:id” element= {<User />} />

<Route path=”/about” element= {<Navigate to=”/contact” />} />

<Routes>

</BrowserRouter>

1. No active Module:

<NavLink to="/home?name=shuvo"

className={ (value)=>value.isActive ? classes.active : “ “ }

> layout

</NavLink>

1. Redirection (Navigation):

# Route Hooks

**useHistory:** The useHistory hook gives you access to the history instance that you may use to navigate.

Example:

import { useHistory } from "react-router-dom";

function HomeButton() {

let history = useHistory();

function handleClick() {

history.push("/home");

}

return (

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

Go home

</button>

);

}

**useLocation:** The useLocation hook returns the location object that represents the current URL

function usePageViews() {

let location = useLocation();

React.useEffect(() => {

ga.send(["pageview", location.pathname]);

}, [location]);

}

**useParams:** useParams returns an object of key/value pairs of URL parameters

function BlogPost() {

let { slug } = useParams();

return <div>Now showing post {slug}</div>;

}

**useRouteMatch**:

# Data fetch method of react

## Fetch API:

export default function Quote () {

    const [quote, setQote]= useState(null);

    useEffect(()=>{

        const fetchQuote = async() => {

  const res = await fetch('http://api.quotable.io/random');

    const data = await res.json();

    setQote(data);

        }

        fetchQuote();

    },[])

    return (

        <h1> Quote </h1>

        <div> { quote?.\_id } </div>

    )

}

## AXIOS API:

With Async:

export default function Quote() {

    const [quote, setQote]= useState(null);

    useEffect(()=>{

const fetchQuote = async() => {

        const res = await axios.get('http://api.quotable.io/random');

        console.log (res);

         setQote(res.data);

}

fetchQuote();

    },[])

    return (

<h1> Quote </h1>

<div> { quote?.\_id } </div>

    )

}

With promises:

**useEffect(()=>{**

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

**.then(res=>{**

**console.log (res);**

**setPost(res.data);**

**})**

**.catch (err=> {**

**console.log (err)**

**})**

**},[id])**