**Index**

1. **create a new React application**
2. **React features**
3. **React Prerequisite -JavaScript Fundamental**
4. **Functional component**
5. **Functional component -props**
6. **Functional component –useState – literal and for object**
7. **Class Component**
8. **Class Component props and state**
9. **Class component t lifecycle**
10. **UseEffect Hooks**
11. **create a new React application**

To create a new React application, you can use the create-react-app command-line tool. Here's the step-by-step process:

First, make sure you have Node.js and npm installed on your machine. You can check this by running node -v and npm -v in your terminal. If these commands return a version number, you're good to go. If not, you'll need to install Node.js and npm first.

Install create-react-app globally on your machine. This is a package provided by Facebook to help bootstrap new React applications.

Use create-react-app to create a new React application.

# Step 1: Check Node.js and npm versions

node -v

npm -v

# Step 2: Install create-react-app

npm install -g create-react-app

# Step 3: Create a new React application

npx create-react-app my-app

Replace my-app with the name you want for your new application. After running these commands, you'll have a new directory with the same name as your application, filled with a basic React application structure. You can start the application by navigating into the directory and running npm start.

**2- React features**

1. Hot Reload

2.

1. **React Prerequisite -JavaScript Fundamental**

* Function Declarations and Arrow Functions
* Template Literals
* Short Conditionals: &&, ||, Ternary Operator
* Array Methods: .map(), .filter(), .reduce()
* Object Tricks: Property Shorthand, Destructuring, Spread Operator
* Promises + Async/Await Syntax
* ES Modules + Import / Export syntax
* Spread Operator + Rest Parameters
* Optional Chaining + Nullish Coalescing

**1. Function Declarations and Arrow Functions**

The basis of any React application is the component. In React, components are defined with both JavaScript functions. In React components return JSX elements .

// JavaScript function: returns any valid JavaScript type

function javascriptFunction() {

return "Hello world";

}

// React function component: returns JSX

function ReactComponent(props) {

return <h1>{props.content}</h1>;

}

==> Arrow function

// Arrow function syntax

const MyComponent = (props) => {

return <div>{props.content}</div>;

};

// Arrow function syntax (shorthand)

const MyComponent = (props) => <div>{props.content}</div>;

/\*

In the last example we are using several shorthands that arrow functions allow:

1. No parentheses around a single parameter

2. Implicit return (as compared to using the "return" keyword)

3. No curly braces for function body

\*/

**2. 2. Template Literals**

With the addition of ES6, we were given a newer form of string called a template literal,

which consists of two back ticks `` instead of single or double quotes.

Instead of having to use the + operator,

we can connect strings by putting a JavaScript expression within a special ${} syntax:

xpression (such as a variable) within a special ${} syntax:

/\*

Concatenating strings prior to ES6.Notice the awkward space after the word Hello?

\*/

function sayHello(text) {

return "Hello " + text + "!";

}

sayHello("React"); // Hello React!

/\*

Concatenating strings using template literals.

See how much more readable and predictable this code is?

\*/

function sayHelloAgain(text) {

return `Hello again, ${text}!`;

}

sayHelloAgain("React"); // Hello again, React!

**2.3. Short Conditionals: &&, ||, Ternary Operator**

conditionally show (or hide) JSX elements using simple if statements .

const isLoggedIn = true;

if (isLoggedIn) {

return <div>Welcome back!</div>;

}

return <div>Who are you?</div>;

using Ternary operator

return isLoggedIn ? <div>Welcome back!</div> : <div>Who are you?</div>;

**2.4 Three Array Methods: .map(), .filter(), .reduce()**

map

const programmers = ["Reed", "John", "Jane"];

return (

<ul>

{programmers.map((programmer) => (

<li>{programmer}</li>

))}

</ul>

);

**JSX ( Js + html ) in depth**

**alternative of React.CreateElement you don’t need to import React for jsx**

**HTML =>** <h1 class =”abc”>

**JSX =>** <h1 className =”abc”>

React 18

1. Dom Rendering

ReactDOM.render() is no longer supported . However it work better to switch CreateRoot() API .



**Why are Components Important in React?**

So, why are components such a big deal in React? There are a few reasons:

**Modularity and Reusability**

By breaking down your UI into components, you create a modular structure. This means you can develop, test, and maintain each piece of your app separately. Plus, once you've created a component, you can reuse it throughout your app, saving you time and effort.

### Efficiency in Development

Components encourage a more efficient development process. You can have different team members working on different components simultaneously without stepping on each other's toes. This speeds up the development process and encourages collaboration.

### Maintainability

Imagine you need to update the styling of a button that appears in multiple places in your app. With components, you only need to update the styling in one place, the button's component, and it will automatically reflect wherever it's used.

## Types of React Components

There are two main types of components in React: functional components and class components.

### functional components

This is the simplest way to define components in React. They are basically JavaScript functions that take in **props** (input data) and return **JSX** (JavaScript Syntax Extension) elements.

const MyComponent = () => {

    return (

      <div>

        <h1>Hello, World!</h1>

      </div>

    );

  };

  export default MyComponent;

Functional component with props [**Pass Data to Components using Props]**

function PersonInfo(props: { name: string; age: number }): JSX.Element {

  return (

    <div>

      <p>Name: {props.name}</p>

      <p>Age: {props.age}</p>

    </div>

  );

}

export default PersonInfo;

call from parent (app.tsx) **render with props**

<PersonInfo name="John" age={25} />

<PersonInfo name="Jane" age={22} />

**Props destructing**

Destructuring was introduced in ES6. It’s a JavaScript feature that allows us to extract multiple pieces of data from an array or object and assign them to their own variables.

const person = {

  firstName: "Lindsay",

  lastName: "Criswell",

  city: "NYC"

}

Before ES6, you had to access each property individually:

console.log(person.firstName) // Lindsay

console.log(person.lastName) // Criswell

console.log(person.city) // NYC

const { firstName, lastName, city } = person;

is equivalent to

const firstName = person.firstName

const lastName = person.lastName

const city = person.city

<div className = "App">

                <Greet active="KAPIL GARG" activeStatus = "CSE"/>

    </div>

const Greet = props =>{

    // Destructuring

    const {active, activeStatus} = props;

    return (

        <div>

              <h3> {active} </h3>

              <h1>{activeStatus}</h1>

        </div>

        )

  }

**Concept of state and useState() hook**

Without useState() react will not track the variable state and updated value will not reflect in dom .

export default function Counter() {

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

  function handleClick() {

    setCount(count + 1);

  }

  return (

    <button onClick={handleClick}>

      You pressed me {count} times

    </button>

  );

Update for obj literal

import React, { useState } from "react";

const CounterwithObjLitral = () => {

  const [state, setState] = useState({ cnt: 0, flag: false });

  const incrementCount = () => {

    setState((prevState) => ({

      ...prevState,

      cnt: prevState.cnt + 1,

    }));

  };

  const decrementCount = () => {

    setState((prevState) => ({

      ...prevState,

      cnt: prevState.cnt - 1,

    }));

  };

  const toggleFlag = () => {

    setState((prevState) => ({

      ...prevState,

      flag: !prevState.flag,

    }));

  };

  return (

    <div>

      <p>Count: {state.cnt}</p>

      <p>Flag: {state.flag ? "true" : "false"}</p>

      <button onClick={incrementCount}>Increment Count</button>

      <button onClick={decrementCount}>decrement Count</button>

      <button onClick={toggleFlag}>Toggle Flag</button>

    </div>

  );

};

export default CounterwithObjLitral;

**Lazy initial state/ load**

Normal

export default function Counter() {

  function initialState() {

    console.log('Initial state called -Expensive process' + Date.now());

    return 0;

  }

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

  function handleClick() {

    setCount(count + 1);

  }

  return (

    <button onClick={handleClick}>

      You pressed me {count} times

    </button>

  );

}

With Lazy initialization / load

export default function Counter() {

  function initialState() {

    console.log('Initial state called -Expensive process' + Date.now());

    return 0;

  }

  const [count, setCount] = useState(()=>initialState());

  function handleClick() {

    setCount(count + 1);

  }

  return (

    <button onClick={handleClick}>

      You pressed me {count} times

    </button>

  );

**Basic event Handling & Parameter passing**

Use fat arrow or return function rather calling

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

      <button onClick={() => setCount(count - 1)}>Decrement</button>

**Apply internal and external css**

**External Stylesheet**

import "./styles.css";

use directly using className

**Inline CSS**

To implement inline CSS, you can create an object containing style references, which can be then called using the style attribute. For example:

const styles = {

  section: {

    fontSize: "18px",

    color: "#292b2c",

    backgroundColor: "#fff",

    padding: "0 20px"

  },

  wrapper: {

    textAlign: "center",

    margin: "0 auto",

    marginTop: "50px"

  }

}

It is then added to an element like this:

<section style={styles.section}>

  <div style={styles.wrapper}>

  </div>

</section>

**Primitive Types & Object Literal with useState()**

**Primitive Types with useState()**

import React, { useState } from "react";

const Counter = () => {

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

  const styles = {

    section: {

      fontSize: "18px",

      color: "#292b2c",

      backgroundColor: "#fff",

      padding: "0 20px",

    },

    wrapper: {

      margin: "0 auto",

      marginTop: "50px",

    },

  };

  // const increment = () => {

  //     setCount(count + 1);

  // };

  // const decrement = () => {

  //     setCount(count - 1);

  // };

  return (

    <>

      <section style={styles.section}>

        <div style={styles.wrapper}>

          <h1>Counter: {count}</h1>

          {/\* <button onClick={increment}>Increment</button>

            <button onClick={decrement}>Decrement</button> \*/}

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

          <button onClick={() => setCount(count - 1)}>Decrement</button>

        </div>

      </section>

    </>

  );

};

export default Counter;

**Object Literal with useState()**

import React, { useState } from "react";

const CounterwithObjLitral = () => {

  const [state, setState] = useState({ cnt: 0, flag: false });

  const incrementCount = () => {

    setState((prevState) => ({

      ...prevState,

      cnt: prevState.cnt + 1,

    }));

  };

  const decrementCount = () => {

    setState((prevState) => ({

      ...prevState,

      cnt: prevState.cnt - 1,

    }));

  };

  const toggleFlag = () => {

    setState((prevState) => ({

      ...prevState,

      flag: !prevState.flag,

    }));

  };

  return (

    <div>

      <p>Count: {state.cnt}</p>

      <p>Flag: {state.flag ? "true" : "false"}</p>

      <button onClick={incrementCount}>Increment Count</button>

      <button onClick={decrementCount}>decrement Count</button>

      <button onClick={toggleFlag}>Toggle Flag</button>

    </div>

  );

};

export default CounterwithObjLitral;

**Class component**

The React.Component needs to extended to create a class component

Instead of return a render() method .

**Class Component state**

import React, { Component } from "react";

class CounterClass extends Component {

  state: { counter: number } = {

    counter: 0,

  };

  increment = () => {

    let c = this.state.counter;

    c++;

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

      return { counter: c };

    });

  };

  decrement = () => {

    let c = this.state.counter;

    c--;

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

      return { counter: c };

    });

  };

  render() {

    return (

      <div>

        <h2>Count: {this.state.counter}</h2>

        <button onClick={this.increment}>Increment</button>

        <button onClick={this.decrement}>Decrement</button>

      </div>

    );

  }

}

export default CounterClass;

**Class Component props**

type CounterState = {

  count: number;

};

type CounterProps = {

  name: string;

};

class CounterWithProps extends React.Component<CounterProps, CounterState> {

  constructor(props: any) {

    super(props);

    console.log("props", props);

    this.state = {

      count: 0,

    };

  }

  handleClick = () => {

    this.setState((prevState: { count: number }, props: CounterProps) => ({

      count: prevState.count + 1,

    }));

  };

  render() {

    return (

      <div>

        <h1>Hello, {this.props.name}!</h1>

        <p>Count: {this.state.count}</p>

        <button onClick={this.handleClick}>Increment</button>

      </div>

    );

  }

}

export default CounterWithProps;

**Component life cycle methods**

|  |  |
| --- | --- |
| **Load/Create** |  |
| **Unload/Cleanup** |  |

**Sure, let's break down the React component lifecycle into simple terms with some easy-to-understand examples:**

**Mounting Phase**:

* **constructor()**: Think of it like setting up your room before guests arrive. You're getting things in order.

javascript

constructor(props) {

super(props);

this.state = { count: 0 };

}

* **render()**: It's like decorating your room with furniture and making it look presentable.

javascript

render() {

return <div>{this.state.count}</div>;

}

* + **componentDidMount()**: Similar to welcoming guests when they arrive at your home. It's triggered once after the component is rendered for the first time.

javascript

componentDidMount() {

console.log('Component mounted!');

}

The **componentDidMount()** method is called once the component has been mounted into the DOM. It is typically used to set up any necessary event listeners or timers, perform any necessary API calls or data fetching, and perform other initialization tasks that require access to the browser's DOM API.

1. **Updating Phase**:
   * **componentDidUpdate()**: This is like when your guests are already there, and you update them with any changes or information.

javascript

componentDidUpdate(prevProps, prevState) {

if (prevState.count !== this.state.count) {

console.log('Count updated!');

}

}

But this method is not recommended for updating the state, as it can cause an infinite loop of rendering. It is primarily used for tasks such as making API calls, updating the DOM, or preparing the component to receive new data **componentWillUpdate**()  is often used in conjunction with  **componentDidUpdate**()  to handle component updates.

* + **shouldComponentUpdate()**: You're deciding whether to update something in your room before your guests see it.

javascript

shouldComponentUpdate(nextProps, nextState) {

return nextState.count !== this.state.count;

}

1. **Unmounting Phase**:
   * **componentWillUnmount()**: When your guests leave, you clean up the room and say goodbye.

javascript

componentWillUnmount() {

console.log('Component will unmount!');

}

**UseEffect Hooks**

**if you know class component lifecycle method you can think of useEffect() hook as componentDidMount(),componentDidUpdate() and componentWillUnmount() combined .**

**Mutations ,subscriptions ,timer,logging and other side effect are not allowed inside the main body of functional component .so useEffect hook is suitable to implement these side effects .**

The useEffect hook in React is used to perform side effects in function components. Side effects are actions that happen outside of the usual flow of your application, such as data fetching, subscriptions, or manually changing the DOM.

Here's a simple breakdown:

1. **What it does**:
   * useEffect allows you to perform side effects in your function components.
2. **When it runs**:
   * It runs after every render by default, including the first render.
3. **Why it's useful**:
   * You can use it to fetch data, set up subscriptions, or perform other side effects.
4. **How to use it**:
   * You pass a function to useEffect that contains the code for your side effect.
5. **Dependencies**:
   * You can also specify dependencies, which are variables that, when changed, should trigger the effect to run again.
6. **Clean-up**:
   * Optionally, you can return a function from the effect, which will be run before the component is unmounted or before the effect runs again.

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

function MyComponent() {

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

    useEffect(() => {

        // This runs after every render

        fetchData();

        // Optionally, return a cleanup function

        return () => {

            // Clean-up code here (if needed)

        };

    }, [/\* dependencies \*/]);

    async function fetchData() {

        const response = await fetch('https://api.example.com/data');

        const newData = await response.json();

        setData(newData);

    }

    return (

        <div>

            {data ? <p>{data}</p> : <p>Loading...</p>}

        </div>

    );

}

In this example:

* We're fetching data from an API after each render using **useEffect**.
* We specify an empty dependency array (**[]**) to indicate that this effect doesn't depend on any variables, so it only runs once (similar to **componentDidMount** in class components).
* If you have dependencies, you list them inside the dependency array, and the effect will rerun whenever any of those dependencies change.

**Digital clock component with useEffect() Hook**

const DigitalClock = (): JSX.Element => {

  const [time, setTime] = useState(new Date());

  useEffect(() => {

    const timer = setInterval(() => {

      setTime(new Date());

    }, 1000);

    return () => {

      clearInterval(timer);

    };

  }, []);

  return (

    <div>

      <h1>{time.toLocaleTimeString()}</h1>

    </div>

  );

};

export default DigitalClock;

**UseEffect() with multiple state (skip or run for certain states)**

useEffect( ()=>{

    //statements

} ,[state1,state2,……stateN] )

**So in dependency of state if changes only then useEffect code run .**

**Condition Rendering (componentWillUnmount)**

**Call this component conditional when component unloaded return() function will run**

useEffect( ()=>{

    //code

return()=>{

    //clean up code

    }

})

**Ref- the older way**

**“ref” the reference of any DOM element using the React .**

In React, a ref is like a "reference" to a specific element in the DOM (Document Object Model), or to a component instance. Think of it as a way to directly access or interact with a particular element or component in your React application.

1. **What it does:**
   * A ref allows you to access or interact with a specific element or component directly.
2. **Why it's useful:**
   * It's useful when you need to focus on an input field, measure the size or position of an element, or trigger imperative animations or actions.
3. **How to use it:**
   * You create a ref using React.createRef() or the useRef() hook.
   * You attach the ref to a React element using the ref attribute.
   * Then, you can access the element or component using the ref's current property.

function MyComponent() {

    const inputRef = useRef(null);

    function focusInput() {

        inputRef.current.focus();

    }

    return (

        <div>

            <input ref={inputRef} type="text" />

            <button onClick={focusInput}>Focus Input</button>

        </div>

    );

}

**list of common use cases for using refs in React:**

1. **Managing Focus:**
   * Auto-focusing on an input field when a component mounts.
   * Setting focus on a specific element when navigating to a new section of a page.
2. **Accessing DOM Elements:**
   * Integrating with third-party libraries that require direct access to DOM elements.
   * Manipulating the DOM directly for specific UI interactions or animations.
3. **Implementing Imperative Animations:**
   * Controlling animations imperatively, such as triggering animations on scroll or hover.
   * Animating elements that are not directly controlled by React's state or props.
4. **Interacting with Embedded Media:**
   * Accessing and controlling embedded media players, such as video or audio players.
   * Implementing custom controls or interactions for embedded media elements.
5. **Integrating with External APIs:**
   * Interacting with external APIs or libraries that require direct access to DOM elements.
   * Managing interactions with browser APIs like localStorage, navigator, etc.
6. **Handling Form Inputs:**
   * Accessing form input values directly for form validation or submission.
   * Implementing custom form controls that require direct interaction with the DOM.
7. **Managing Third-Party Components:**
   * Integrating and controlling third-party UI components that require direct access to DOM elements.
   * Implementing custom behavior or interactions with third-party components.
8. **Implementing Custom Hooks:**
   * Building custom hooks that encapsulate complex logic and may require access to DOM elements.
   * Managing state or behavior that spans multiple components using refs within custom hooks.
9. **Scrolling Behavior:**
   * Implementing custom scrolling behavior, such as smooth scrolling or infinite scrolling.
   * Handling scroll-based animations or effects.
10. **Managing Selections and Cursors:**
    * Implementing custom text selection behavior or managing cursor positions in text editors.
    * Building custom selection controls or interactions.