**REACT**

**1. Introduction to React**

React is a JavaScript library for building user interfaces. It allows you to create reusable UI components that manage their own state.

**Setting Up a React Project**

To start with React, you typically use Create React App:

npx create-react-app my-app

cd my-app

npm start

This sets up a new React project with all the necessary build tools and configurations.

### 2. JSX (JavaScript XML)

JSX is a syntax extension for JavaScript that allows you to write HTML-like code inside your JavaScript files. It makes writing React components easier.

import React from 'react';

function HelloWorld() {

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

}

export default HelloWorld;

### 3. Components

React applications are built using components. Components can be functional or class-based.

#### Functional Components

Functional components are the simplest way to create components:

import React from 'react';

function Welcome(props) {

return <h1>Hello, {props.name}</h1>;

}

export default Welcome;

#### Class Components

Class components are more powerful and can have their own state and lifecycle methods:

import React, { Component } from 'react';

class Welcome extends Component {

render() {

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

}

}

export default Welcome;

### 4. Props

Props (short for properties) are how you pass data from parent to child components.

import React from 'react';

function Welcome(props) {

return <h1>Hello, {props.name}</h1>;

}

function App() {

return (

<div>

<Welcome name="Alice" />

<Welcome name="Bob" />

</div>

);

}

export default App;

### 5. State

State is how you manage data that changes over time within a component.

#### Using State in Class Components

import React, { Component } from 'react';

class Counter extends Component {

constructor(props) {

super(props);

this.state = { count: 0 };

}

increment = () => {

this.setState({ count: this.state.count + 1 });//we pass the partial state we want to update

};

render() {

return (

<div>

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

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

</div>

);

}

}

export default Counter;

#### Using State in Functional Components with Hooks

Hooks allow you to use state and other React features in functional components.

import React, { useState } from 'react';

function Counter() {

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

return (

<div>

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

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

</div>

);

}

export default Counter;

**6. Lifecycle Methods**

Lifecycle methods are special methods in class components that run at specific points in a component's life.

**Common Lifecycle Methods**

* **componentDidMount()**: Runs after the component is first rendered.
* **componentDidUpdate()**: Runs after the component is updated.
* **componentWillUnmount()**: Runs just before the component is removed from the DOM.

import React, { Component } from 'react';

class MyComponent extends Component {

constructor(props) {

super(props);

this.state = {

count: 0

};

console.log('Constructor');

}

componentDidMount() {

console.log('Component Did Mount');

}

componentDidUpdate(prevProps, prevState) {

console.log('Component Did Update');

console.log('Previous State:', prevState);

console.log('Current State:', this.state);

}

componentWillUnmount() {

console.log('Component Will Unmount');

}

incrementCount = () => {

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

}

render() {

console.log('Render');

return (

<div>

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

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

</div>

);

}

}

export default MyComponent;**7. Hooks**

Hooks are functions that let you use state and other React features in functional components.

USEEFFECT:

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

function MyComponent() {

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

useEffect(() => {

// This runs after every render

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

// This is optional cleanup code

return () => {

document.title = 'React App';

};

}, [count]); // Only re-run the effect if count changes

return (

<div>

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

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

Click me

</button>

</div>

);

}

In this example:

* We have a counter **count**.
* We use **useEffect** to update the document title with the count after every render. We also clean up this effect when the component unmounts.
* The second argument **[count]** tells React to re-run the effect only when **count** changes.

So, **useEffect** lets you do things in React components that happen after the component is rendered, like updating the document title, fetching data from a server, or setting up subscriptions. It's a way to add more functionality to your components.

In this **useEffect** hook:

* The effect function inside **useEffect** runs after every render of the component.
* However, the effect function updates the document title based on the current value of the **count** state variable.
* The effect function will run again only if the **count** state variable changes. This is specified by **[count]** as the dependency array.

**document** refers to the Document Object Model (DOM) in JavaScript. It represents the web page loaded in the browser and provides methods and properties for interacting with the elements on the page.

In the context of React:

* **document** is a global object that represents the entire HTML document.
* You can use **document** to access and manipulate elements on the web page, such as changing the title, modifying styles, adding or removing elements, handling events, etc.

**Common Hooks**

* **useState()**: Manages state in a functional component.
* **useEffect()**: Performs side effects in function components.
* **useContext()**: Accesses context values in functional components.
* import React, { useState, useEffect } from 'react';
* function Timer() {
* const [count, setCount] = useState(0);
* useEffect(() => {
* // This function runs after every render
* const interval = setInterval(() => {
* // Update the count state variable every 1000 milliseconds (1 second)
* setCount(c => c + 1);
* }, 1000);
* // This function runs when the component unmounts or when the dependency array is updated
* return () => {
* // Cleanup: Clear the interval to prevent memory leaks
* clearInterval(interval);
* };
* }, []); // Empty dependency array indicates that this effect has no dependencies
* // It runs only once after the initial render
* return <div>Timer: {count}</div>; // Render the current count
* }
* export default Timer;

IMP TERMS

**Rendering:**

* **Rendering** in React refers to the process of generating the output (typically HTML) that the user will see on the screen based on the component's current state and props.
* When a component is **rendered**, React creates a virtual representation of the component's UI (Virtual DOM) and updates the actual DOM (Document Object Model) to reflect the changes.
* React components are rendered as HTML elements in the DOM, allowing users to see and interact with them in the web browser.

**Mounting:**

* **Mounting** is the initial process of creating and inserting a component into the DOM. It happens when a component is first rendered.
* During mounting, React creates the component's DOM nodes and inserts them into the DOM tree, making the component and its children visible on the screen.
* Lifecycle methods like **componentDidMount** (for class components) or the **useEffect** hook (for function components) are called during mounting, allowing you to perform initialization tasks, data fetching, or setup operations.

**Unmounting:**

* **Unmounting** is the process of removing a component from the DOM. It happens when a component is no longer needed or when it's removed from the render tree.
* During unmounting, React removes the component's DOM nodes from the page, effectively "unrendering" it and cleaning up any resources associated with it.
* Lifecycle methods like **componentWillUnmount** (for class components) or the cleanup function in the **useEffect** hook (for function components) are called during unmounting, allowing you to perform cleanup tasks like clearing timers, unsubscribing from event listeners, or releasing resources.

In summary, when a component is **rendered**, it's added to the DOM, making it visible on the screen. When a component is **unmounted**, it's removed from the DOM, cleaning up any resources and effectively removing it from the user interface. These processes are fundamental to how React manages components and updates the user interface in web applications.

### 8. Context

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

import React, { createContext, useContext } from 'react';

const MyContext = createContext();

function MyProvider({ children }) {

const value = 'Hello, Context!';

return <MyContext.Provider value={value}>{children}</MyContext.Provider>;

}

function MyComponent() {

const value = useContext(MyContext);

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

}

function App() {

return (

<MyProvider>

<MyComponent />

</MyProvider>

);

}

export default App;

No worries! Let's clarify what a component tree is:

**Component Tree in React:**

* In React, a **component tree** refers to the hierarchical structure of components in a React application.
* A component tree represents the relationships between components, with each component being a node in the tree.
* Components can be nested within other components, forming parent-child relationships, and creating a tree-like structure.
* The root of the component tree is typically the top-level component rendered by ReactDOM.

(IN JS FUNCTIONS/REACT COMPONENTS WHEN THE PARAMETER IS ENCLOSED WITHIN BRACES IT MEANS WE’RE DESTRUCTURING THAT KEY)

**CREATING REDUX STORE**

import { createStore } from 'redux';

const initialState = {

value: 0,

};

function counterReducer(state = initialState, action) {

switch (action.type) {

case 'increment':

return { value: state.value + 1 };

case 'decrement':

return { value: state.value - 1 };

default:

return state;

}

}

const store = createStore(counterReducer);

export default store;

Connecting Redux with React:

import React from 'react';

import ReactDOM from 'react-dom';

import { Provider, useSelector, useDispatch } from 'react-redux';

import store from './store';

function Counter() {

const count = useSelector((state) => state.value);

const dispatch = useDispatch();

return (

<div>

<p>{count}</p>

<button onClick={() => dispatch({ type: 'increment' })}>Increment</button>

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

//dispatch automatically dispatches to redux store

</div>

);

}

ReactDOM.render(

<Provider store={store}>

<Counter />

</Provider>,

document.getElementById('root')

);

This line of code, const count = useSelector((state) => state.value);, is using React Redux's useSelector hook to access a value from the Redux store in a React functional component. Here's a breakdown of what it does:

**1. useSelector Hook:**

* This hook is provided by the react-redux library.
* It allows React functional components to access data stored in the Redux application state.

**2. Selector Function:**

* The useSelector hook takes a callback function (also called a selector) as an argument. This function is responsible for selecting a specific piece of data from the entire Redux state object.

**3. (state) => state.value:**

* This is the selector function passed to useSelector. It receives the entire Redux state object (state) as an argument.
* Inside the function, it simply returns state.value. This assumes there's a property named value directly on the root level of your Redux state object.

**4. Assigning the Value:**

* The result of the selector function (which in this case is state.value) is assigned to the constant count.

**Configure Redux Persist**:

import { createStore } from 'redux';

import { persistStore, persistReducer } from 'redux-persist';

import storage from 'redux-persist/lib/storage'; // defaults to localStorage for web

import { Provider } from 'react-redux';

import { PersistGate } from 'redux-persist/integration/react';

import React from 'react';

import ReactDOM from 'react-dom';

const persistConfig = {

key: 'root',

storage,

};

const persistedReducer = persistReducer(persistConfig, counterReducer);

const store = createStore(persistedReducer);

const persistor = persistStore(store);

function App() {

return (

<Provider store={store}>

<PersistGate loading={null} persistor={persistor}>

<Counter />

</PersistGate>

</Provider>

);

}

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