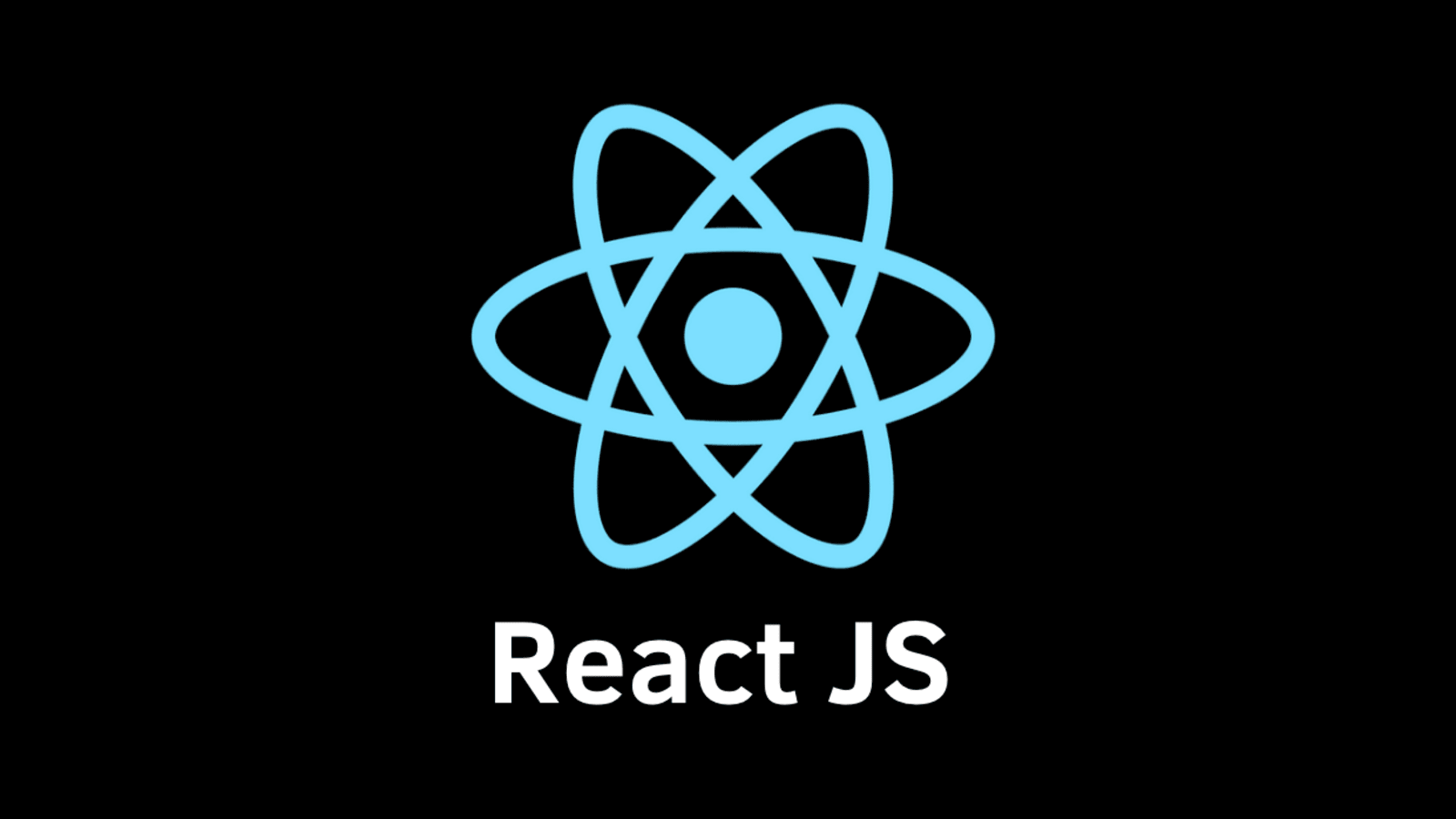
INTRO TO REACTJS



What is ReactJs?

React is a JavaScript library for building SPAs with interactive user interfaces.

The entire React application can be modeled as a set of independent, isolated and reusable components. In other words you can say components are the building blocks of React App. These components are put together to design complex layouts.

React makes it painless to create interactive UIs. It has a concept called state for this purpose.

Some of the websites built with React - Facebook, Netflix, Instagram, Twitter, Airbnb, Nike, MakeMyTrip, Codepen, Snapchat, Prime Videos and many many more.

Single Page Applications

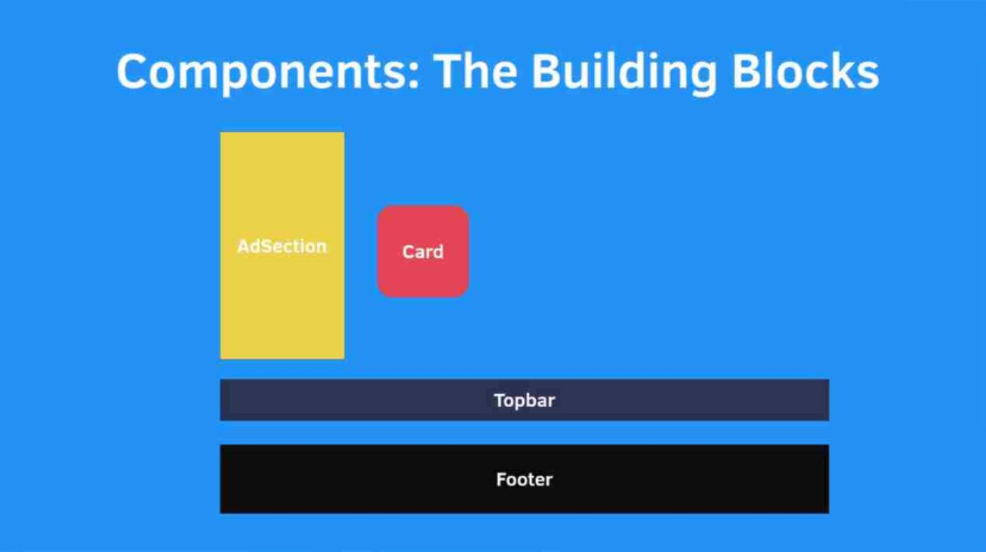
A single-page application is an app that renders inside a browser and does not require page reloading during use. For eg, Gmail, Facebook, Prime Videos, Trello and many more.

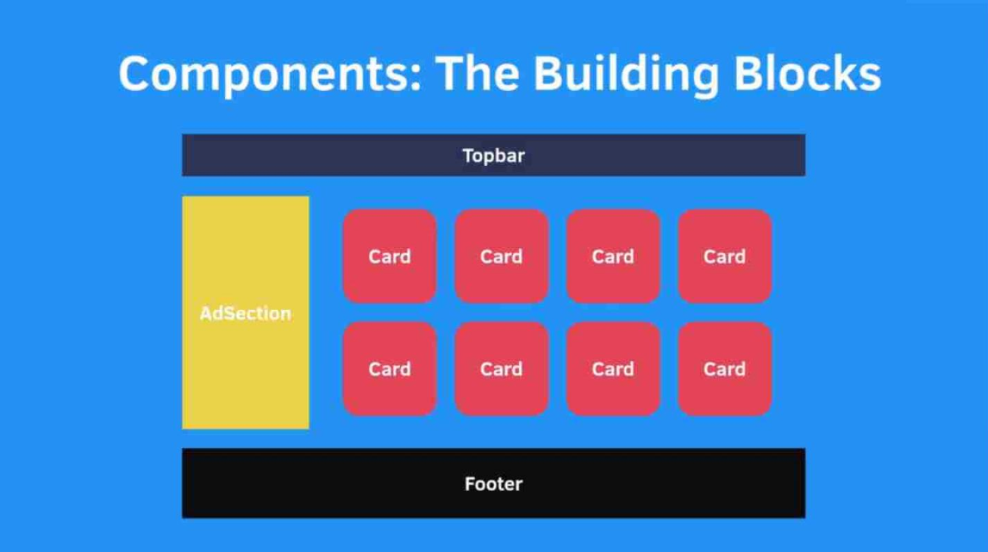
When you open a SPA in browser. It brings all the content in the first load itself. Now whenever you change routes or click on dropdowns it doesn’t needs to refresh the page to fetch data because all the required data is already present.

Some latest Frontend Frameworks like React, Angular, Vue, Ember etc enable us to create SPAs.

Apps built using react







To use React we need two things installed –

1. Node
2. Text Editor of your choice

Let’s Create a **React** Project

* If you want to build a react application there are a lots of libraries which are required to make the application run properly

Ex : React , React-dom, webpack, babel etc .

This leaves us with 2 options

1. First is to Install all the libraries and configure the entire project ourself
2. We can use a tool provided by React-developers. It is called **Create-React-App**.It installs all the libraries required and creates a configuration for our project . It also adds some stater files.

Hello World App!!

Create React App is an officially supported way to create single-page React

applications. It offers a modern build setup with no configuration.

You’ll need to have Node >= 8.10 on your local development machine.

To create a new app, follow these steps:

node -v (version), npm -v()

npx create-react-app my-app

cd my-app

npm start

**Npm** - It is the Node package manager. It will help you Install and manage the packages and libraries required for your applicaton. It is installed automatically when you Install Node.js .

**Npx** – It is nothing but a Node package runner.

React Project Structure

**Node-modules** - It contains all the dependencies or packages installed by Create-React-app to run your application. It is generated when you run Create-react-app command or when you run Npm Install.

**Manifest.JSON** – This object contains details of your application .if your building a progressive web app.

A progressive web application, or progressive web app, is a type of application software delivered through the web, built using common web technologies including HTML, CSS, JavaScript

**ReportWebVitals.js –** It gives you the Performance report.

**SetupTests.js –** Test cases setup starts from here.

**Package.json** – All the dependencies and libraries required to run the project are listed here.

**Package-lock.JSON –** It keeps tracks of exact version of every package that is Installed,It contains more details about the package . This ensures that the same versions of dependencies are used consistently across different environments and installations. It helps in preventing unexpected changes in your project's dependencies.

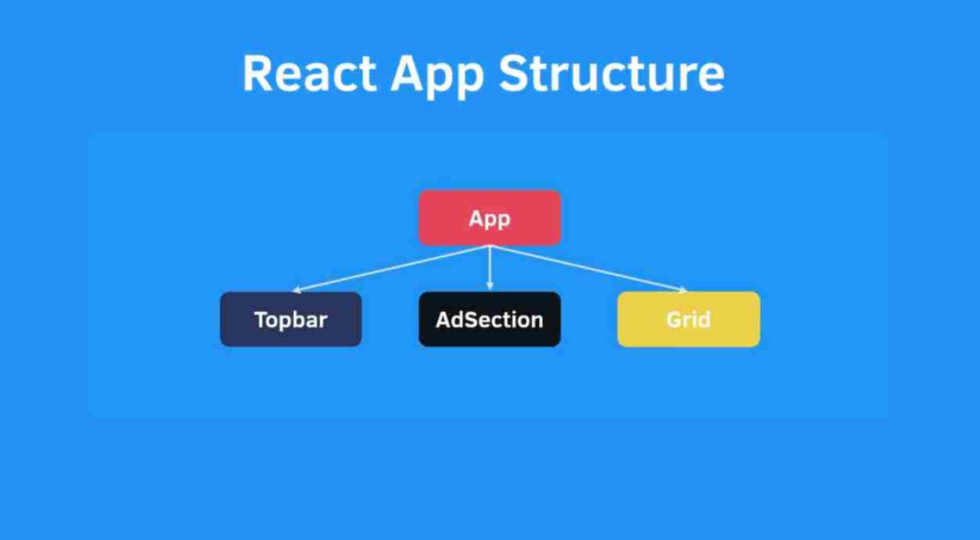
**App.test.js** – Test cases are wriiten in this file .

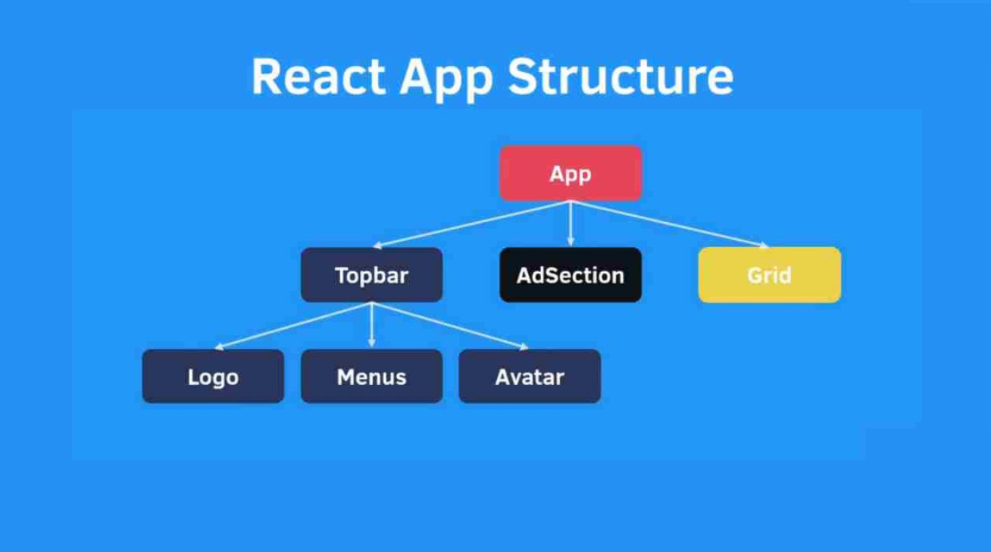
**App.js** – We start writing code from the file

**Index.js –** Index.js is the entry point.









Understanding JSX

This funny tag syntax is neither a string nor HTML. It is called JSX, and it is a syntax extension to JavaScript.

It is used to describe what the UI should look like. It brings the HTML markup and the display logic together. It couples the rendering logic with other UI logic.There are some {} in this JSX code. Inside these curly brackets you can write JavaScriptexpressions (which generate some value by default)

For example,

const element = <span>Hello {name}!!</span>

JSX = Write Lesser Code

For eg, Adding Elements Dynamically.

**HTML**

<ol id="demo"></ol>

**Vanilla JavaScript**

var firstName = “John”

var list= document.getElementById(demo).value;

var entry = document.createElement('li');

entry.appendChild(document.createTextNode(firstName));

list.appendChild(entry);

**JSX**

<ol><li>{firstname}</li></ol>

JSX Basics

1. Embed Expressions.

For eg,

const greetings = <h1>Welcome, {firstname}</h1>

fullName = (firstName, lastName) => {

return firstName + lastName

}

const greetings = <h1>Welcome, {fullName(“John”, “Lark”)} </h1>

2. HTML Attributes.

For eg,

<a className=”GoogleLink” href=”https://www.google.com” >Link</a>

3. Dynamic Attributes.

For eg,

const profilePic = <img src={user.profilePic} alt=”Profile Pic” />

React Components

To use JSX we need to import the ‘React’ module from the installed ‘react’ package.

import React from ‘react’;

const MenuItem = () => {

return <span>Hello React</span>

}

export default MenuItem;

React.createElement()

We can also create elements using the React.createElement() method.

When React creates component/element, it calls this method, which takes three arguments.

- The component/element name.

- An object representing the element’s properties also called props.

- An array of the element’s children.

Babel

Babel us a tool that allows you to write code that might not work in the browser. Babel is going to do the hard work of compiling your code to a format that is compatible with the browser .

Create Component from Lists

We can use map to iterate list items and generate dynamic HTML elements.

const blogs = [{title:’’, desc:’’}, {title:’’, desc:’’}, ...]

const blogList = blogs.map((item, pos) => {

return <div key={pos}> <h3>{item.title}</h3> <p>{item.desc}</p> </div>

});

Key helps React identify which components have changed. All the elements inside an array should have a unique key.

Adding Inline Styles

To add inline styles you can simply use the styles attribute and give it an object.

For eg,

<div styles={{backgroundColor: ‘blue’, fontSize: ‘16px’}}>Hello<div>

const styles = {

backgroundColor: ‘blue’,

fontSize: ‘16px’

}

<div styles={styles}>Hello<div>

Adding External Styles

To add external styles, just need to import the stylesheets file.

To select JSX elements use the className property.

For eg,

import ‘./App.css’;

<h3 className=”Heading”> Welcome to React </h3>

.Heading {

color: red;

}

Functional Component

The simplest way to define a component is to write a JavaScript function.

For eg,

const blogList = () => {

return() {

<div>

<h3>Blog 1</h3>

<p>Blog Desc 1</p>

</div>

}

}

Modules Export and Import

ES6 enables developers to write modular code. Basically, we can split reusable code into separate files.

Export makes a module accessible in other files. Modules can be exported in two ways

- default and named export.

Import is used to add different modules in a JavaScript file. It has different syntax based on the export method used by the target module.

Component Props

This props is used to pass data to the components. It is an object with some key-value pairs to hold passed data. It is a read-only object.

<BlogItem title={‘---’} desc={‘----’} />

const BlogItem = (props) => {

return (<div>

<h3>{props.title}</h3>

<p>{props.desc}</p>

</div>);

}

React Fragment

In React, a fragment is a component that allows you to group multiple child elements without adding an extra DOM element to the output. Fragments are a way to avoid unnecessary container elements in your component's render output. They are especially useful when you want to return multiple elements from a component's render method without introducing an additional div or other container element in the HTML structure.

Fragments were introduced in React 16.2 as a way to address the issue of adding unnecessary elements to the DOM structure when you return multiple elements from a component. Prior to fragments, you might have had to wrap the child elements in a parent container element, which could affect the layout and styling of your application.

Here's how you can use a fragment in React:

<React.Fragment>

<h1>Hello</h1>

<p>This is a paragraph.</p>

</React.Fragment>

Alternatively, you can use the shorthand syntax for fragments using empty angle brackets <> and </>:

<>

<h1>Hello</h1>

<p>This is a paragraph.</p>

</>

Class-based Component

Class-based components in React are a way to create reusable user interface elements using JavaScript classes.

These components were the primary way to define components in React before the introduction of functional components with hooks.

Class-based components extend the React.Component class and provide a structured and object-oriented approach to building user interfaces.

A class-based component inherits the Component class of React. This Component class gives the component access to some properties and methods like state and setState etc.

This component is a base class for all class based React.js components

Why do we need to extend component class in react ?

In React, when you create a class-based component, you extend the React.Component class to inherit a set of functionalities and behaviors that are essential for building and managing user interface components. Extending the React.Component class provides the following benefits:

* **Lifecycle Methods**: The React.Component class defines a set of lifecycle methods that you can override in your component. These methods allow you to hook into different phases of a component's life, such as component creation, rendering, updating, and destruction. By extending the class, you can implement custom behavior in response to these lifecycle events, like componentDidMount, componentDidUpdate, and componentWillUnmount.
* **State Management**: The React.Component class provides a built-in mechanism for managing a component's state. You can use the this.state property and this.setState() method to manage and update the component's data. State management is essential for building dynamic and interactive user interfaces.
* **Props Handling**: Components receive data from their parent components through props. Extending React.Component allows you to access and use these props via this.props. You can then use the received data to customize the rendering of your component.
* **Reconciliation** and Rendering: The base React.Component class handles the process of reconciling the component's virtual representation (the Virtual DOM) with the actual DOM and performing efficient updates. This helps ensure that only the necessary parts of the DOM are changed when the component's state or props change, improving performance.
* **Component Hierarchy**: Class-based components fit well within the hierarchical structure of React applications. They can be easily composed to create complex UIs by nesting components within one another, and this hierarchy is a fundamental concept in React development.

What is a State ?

The state is a built-in React object that is used to contain data or information about the component. A component's state can change over time; whenever it changes, the component re-renders.

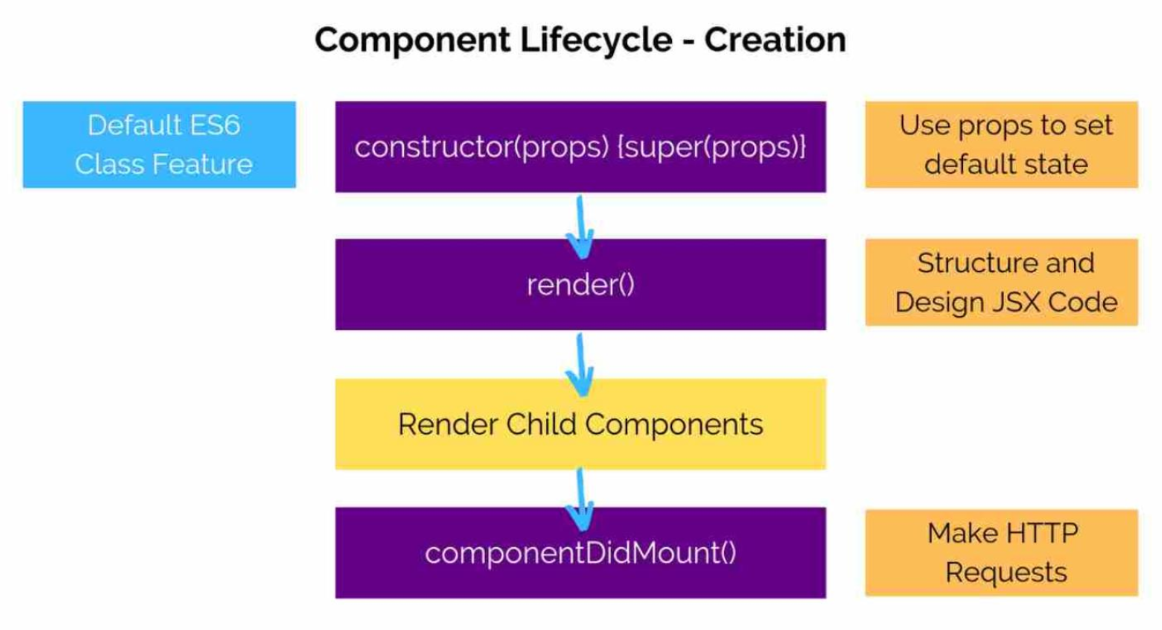
What is setState ?

In React, setState is a method provided by class-based components and some legacy versions of functional components with state (before the introduction of hooks). It is used to update the state of a component. The setState method allows you to modify the component's state in a way that triggers a re-render of the component, reflecting the updated state in the user interface.

Lifecycle methods in react

What a react component is created a component goes through several phases in its lifecycle :

1) **Mounting** : Mounting phase happens when a component is being created and inserted into the Dom



1. **Constructor** : a special function that will get called whenever a new component is created. Two reasons to use constructor in react :

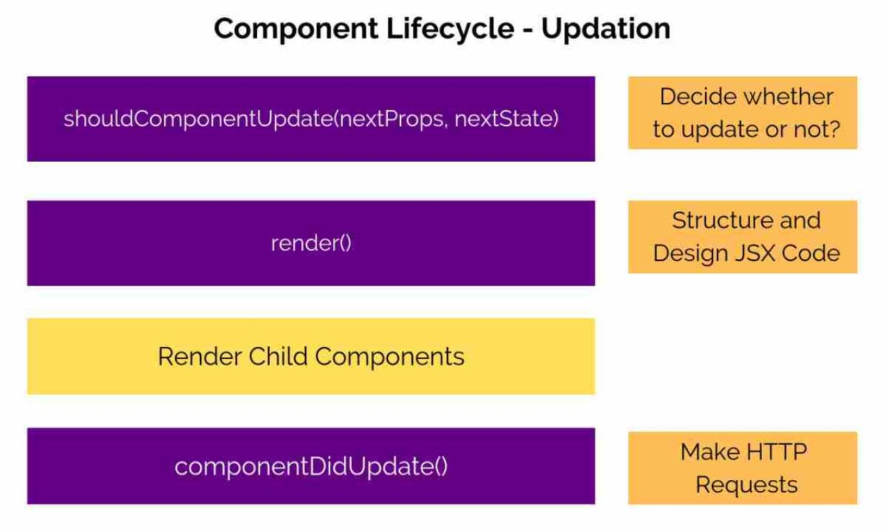
* Initializing state
* Binding event handlers

1. **render()** : only required method in class component . It is responsible for returning the JSX (JavaScript XML) that defines the component's UI. The render method is called whenever the component is initially mounted, and whenever the component's state or props change. It returns the virtual representation of the component's UI, which React uses to update the actual DOM when changes occur.
2. **componentDidMount()** : called immediately after a component and all its children components have been rendered to the Dom**.It is a perfect place to make AJAX Calls** .

Why is componentDidMount perfect place to make api calls ?

* By the time componentDidMount is called, the component has been fully rendered in the DOM. This means that the component's structure and elements are available, making it an appropriate point in the component's lifecycle to interact with the DOM or external resources like APIs.
* Placing API calls in componentDidMount ensures that these calls are only made once, immediately after the component is mounted. This helps prevent unnecessary API requests when a component re-renders due to state or prop changes.

**2) Updating :** The updating phase occurs when a component is being re-rendered as a result of changes in its state or props.



**shouldComponentUpdate() :**  Rendering is a heavy process. Especially when the layout is very complex. Everytime the setState() function is called. It starts the entire cycle of component update. A new virtual DOM is generated, compared with the old version of virtual DOM and then it updates the actual DOM. This can hurt the performance of a React App drastically if done wrong. So, the rule is simple. Keep re-renders as minimum as possible. We can avoid unnecessary re-renders in the shouldComponentUpdate() lifecycle hook.It gives us access to nextProps and nextState. We can compare if the value is changed

or not. If the value is changed then we can re-render or else we can can avoid the re-render.

Syntax: shouldComponentUpdate(nextProps, nextState) {

return true/false;

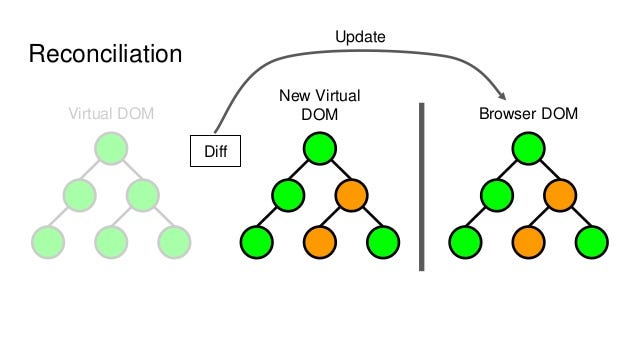
}

1. **Unmounting :** The unmounting phase happens when a component is being removed from the DOM. It includes the **componentWillUnmount()** method, which is called just before the component is unmounted from the DOM.This phase is used for cleanup and teardown operations, such as removing event listeners or canceling asynchronous tasks.

**NOTE** : It's important to note that the lifecycle methods in the mounting and updating phases are executed in a specific order, and not all methods are required in every component. Additionally, with the introduction of React Hooks in functional components, some of the class-based lifecycle methods have been replaced by useEffect and other hooks. When using functional components, the lifecycle phases may be handled differently.

Virtual DOM

* Virtual Dom is a lightweight JavaScript representation of Dom used in React
* Virtual Dom is a copy or virtual representation of a UI
* React has 2 virtual DOM’s . suppose there is a change in state then old virtual dom represents Dom without changes , new virtual Dom represent the changes . Now Both the virtual Dom’s are compared and only the changes will be re-rendered in the actual dom .
* Updating virtual Dom is comparatively faster than updating the actual Dom .



Pure Components

**PureComponent** is another class provided by React. It automatically checks if the state or props has not changed then it will avoid a re-render.It does a shallow comparison of props and states data. This doesn’t gives us a lot of control but works well for cases when the data is simple.

For eg,

import {PureComponent} from ‘react’;

class Person extends PureComponent {

//Component Code

}

Intro to Hooks

In React, "hooks" are functions that allow you to use state and other React features in functional components. They were introduced in React 16.8 to enable functional components to manage state and side effects, previously only possible with class components. Hooks have become a fundamental part of React development. Here are some commonly used hooks:

* **useState**: This hook allows functional components to manage local component state. It takes an initial state value and returns an array with the current state and a function to update it.

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

* **useEffect**: This hook lets you perform side effects in your functional components. It takes a function that contains the code you want to run and an optional array of dependencies that specify when the effect should run.

useEffect(() => {

// Your side effect code here

}, [dependencies]);

React.memo

if you want to prevent a component from re-rendering when the prevState and nextState are the same, you can use the shouldComponentUpdate lifecycle method for class components or the React.memo higher-order component for function components.

import React from 'react';

const MemoizedComponent = React.memo(({ name, age }) => {

  console.log(`Rendering MemoizedComponent for ${name}`);

  return (

    <div>

      <h2>Name: {name}</h2>

      <p>Age: {age}</p>

    </div>

  );

});

const ParentComponent = () => {

  const [person, setPerson] = React.useState({ name: 'John Doe', age: 25 });

  const updatePerson = () => {

    // Simulating a state update that doesn't change the props passed to MemoizedComponent

    setPerson({ ...person, age: person.age + 1 });

  };

  return (

    <div>

      <MemoizedComponent name={person.name} age={person.age} />

      <button onClick={updatePerson}>Update Age</button>

    </div>

  );

};

export default ParentComponent;

Handling Forms

There are two ways to create Forms in React - Uncontrolled Forms and Controlled Forms.

We can create forms using the normal HTML way, this is called uncontrolled form because the state of the form is handled by Dom.Another way is to handle the entire state using React, this is called controlled form.

On submit, we can extract values from the form or fetch it from our state and send it to the backend using an HTTP post call.

A Controlled component is a component that lets react handle form data for it internally whereas an uncontrolled component is one that stores the form data in the DOM.

Benefits of using Controlled component

* When using an uncontrolled component you only access the values of the input when you submit the form ,but when using a controlled component ,we always have access to the value.so it makes easy to implement validation checks on key stroke.
* You can use the component re-render to your benefit
* You can easily update the component values by modifying the state .this is essential for building interactive forms and dynamic UI .
* Controlled components work well with other react components ,making it easier to pass and share data between components.

Setting up React-Router

React is a library and it doesn’t comes with out-of-the-box tools to handle page redirection. For this purpose we have a package called React Router DOM. The React Router DOM package requires another package called React Router.

To install React Router:

1. Search React Router npm in the browser.

2. Open the webpage.

3. Copy the command to install React Router.

4. Open your terminal and run the copied command.

5. Similarly do it for React Router DOM.

BrowserRouter

BrowserRouter is a component provided by the react-router-dom library in React. It's a type of router that uses the HTML5 History API to keep your UI in sync with the URL in the browser. In other words, it enables the navigation capabilities in a React application, allowing you to define different views or components for different routes.

import { BrowserRouter } from 'react-router-dom';

Route

In React Router, you typically use the Route component to define your routes. The Route component allows you to associate a particular path with a component to be rendered when that path is matched.

import { Route } from 'react-router-dom';

<Route exact path="/" component={Home} />

<Route path="/about" component={About} />

<Route path="/contact" component={Contact} />

What are links?

Links are also known as hyperlinks. They are references to data that the user can follow by clicking or tapping. They can be used to view resources on the web. Resources can be web pages, images, videos, etc. When links are clicked, they take you to the resource you requested.

Types of links in React applications

In ReactJS, there are three different kinds of links. These are NavLink, Link, and a links, and they all serve different purposes.

1. **NavLink**: This is used when you want to highlight the current or active link. This is used with the **activeClassName** attribute, which enables it. See the example below.

<NavLink to="/home" activeClassName="active" >Home</NavLink><br/>

The CSS can then be styled according to your choice inside the App.css file. Let’s make the text color red to make it simple.

.active{

  color:red;

}

1. **Link**: This is used when there is no special style or highlighting of your link. See the example below.

<Link to="/not-active">Not Active </Link><br/>

**Note**: Use the NavLink or Link when you need links that are routing to pages that belong to your application. For external links, a is preferrable.

1. **The anchor tag a**: This is used for links outside your webpage.

See the example below.

<a href="https://www.educative.io/answers">Visit Edpresso </a>

Conclusion

The **NavLink** is used when you want to highlight a link as active. So, on every routing to a page, the link is highlighted according to the activeClassName. **Link** is for links that need no highlighting and it doesn’t refresh the page . And **a** is for external links.

Implement Nested Routes in React.js – React Router DOM V6

Resource : https://www.geeksforgeeks.org/implement-nested-routes-in-react-js-react-router-dom-v6/

**Lazy loading**

Lazy loading in React is a technique that defers the loading of certain parts of a web application until they are actually needed. The primary goal of lazy loading is to improve the initial loading time of a web page by loading only the essential resources initially and delaying the loading of additional resources until they are required. This is particularly useful for larger applications where loading all code and assets upfront may result in slower initial page loads.

import React, { lazy, Suspense } from 'react';

// Use React.lazy() to lazily load the component

const MyComponent = lazy(() => import('./MyComponent'));

function App() {

  return (

    <div>

      <h1>Welcome to My App</h1>

      {/\* Use Suspense to handle the loading state \*/}

      <Suspense fallback={<div>Loading...</div>}>

        {/\* Lazily loaded component \*/}

        <MyComponent />

      </Suspense>

    </div>

  );

}

export default App;

In this example:

1. The **MyComponent** is imported using the **React.lazy(() => import('./MyComponent'))** syntax. This indicates that **MyComponent** should be loaded lazily when it is actually needed
2. The **Suspense** component is used to wrap the lazily loaded component. The **fallback** prop specifies what to render while the lazily loaded component is being fetched. In this case, it's a simple "Loading..." message.

Lazy Loading Example with Routing

import React,{lazy,Suspense} from 'react'

import { BrowserRouter,Routes,Route,Link } from 'react-router-dom'

let AboutComponent = lazy(() => import("./About")   )

let HomeComponent = lazy(() => import("./Home")   )

function App() {

  return (

    <>

    <h1>  App Component </h1>

    <BrowserRouter>

    <Link to = "/home" > Home </Link>

    <br></br>

    <Link to = "/about" > About </Link>

    <Routes>

    <Route path = "/home" element = { <Suspense fallback = {<h1> Loading... </h1> } > <HomeComponent /> </Suspense>  }  />

      <Route path = "/about" element = { <Suspense fallback = {<h1> Loading... </h1>}> <AboutComponent /> </Suspense>  }  />

    </Routes>

    </BrowserRouter>

    </>

  )

}

export default App

useLocation() Hook

In React, useLocation is a hook provided by the React Router library. It allows you to access the current location object, which contains information about the current URL.

Here's a basic example of how you might use the useLocation hook:

import React from 'react';

import { useLocation } from 'react-router-dom';

function MyComponent() {

  const location = useLocation();

  return (

    <div>

      <h2>Current Pathname:</h2>

      <p>{location.pathname}</p>

      <h2>Current Search:</h2>

      <p>{location.search}</p>

    </div>

  );

}

In this example, useLocation is used to get the current location object, and we display the pathname and search properties of that object. This can be useful for accessing and manipulating the current URL in a React component.

Make sure that you have the react-router-dom library installed and that your component is rendered within a Router component (usually provided by BrowserRouter or other router components from React Router) for useLocation to work correctly.

PropTypes

"PropTypes" is a feature in React that allows you to specify the types of props that a React component should receive. It helps you catch potential bugs and provide documentation for your components. PropTypes are a way to validate that the data your component receives via props matches the expected types and shapes.

Import the PropTypes library:

import PropTypes from 'prop-types';

**When you are expecting numbers as prop**

App.propTypes = {

Num1 : propTypes.number

}

Other data types which can be checked : array, bool, func , string

**When you are expecting a prop i.e prop is required**

App.propTypes = {

Num1 : propTypes.number.isRequired

}

**Check if prop is renderable**

App.propTypes = {

a : propTypes .node

}

**If prop is required and it can be of any type:**

App.propTypes = {

a : propTypes .any.isRequired

}

**If props should be any one of the type :**

App.propTypes = {

a : propTypes.oneOfType([ propTypes.string , propTypes.number ])

}

**If prop values should only be those words**

App.propTypes = {

a : propTypes.oneOf([ “Loading” , “Ready” ])

}

Context API

The Context API is a part of React that provides a way to share values, such as state or functions, across a tree of React components without having to pass the data explicitly through each level as props. It helps in solving the problem of prop drilling, where you need to pass down data through multiple levels of components, even if some of those components don't use the data themselves.

1. **createContext**: The createContext function is used to create a new context

const MyContext = React.createContext();

1. **<MyContext.Provider>:** The Provider component is used to wrap the part of the component tree where you want to make the context available. It takes a value prop, representing the current value of the context.

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

{/\* Your components go here \*/}

</MyContext.Provider>

1. **<MyContext.Consumer>:** The Consumer component is used to consume the context value within a functional component. It uses a render prop pattern where a function is passed as a child, and that function receives the current context value as an argument.

<MyContext.Consumer>

{value => /\* render something based on the context value \*/}

</MyContext.Consumer>

(OR)

Import {useContext} from “react”

Import {myContext} from “./App”

const value = useContext(myContext);

The Context API is particularly useful in scenarios where you have a need for global state or when passing props through multiple levels becomes impractical. However, for more complex state management or features like middleware, you might consider using state management libraries like Redux.

When to Choose Context API or Redux:

**Use Context API:**

* For simpler state management needs.
* When managing local state within a component or a small subtree of components.
* When the overhead of Redux seems unnecessary for the size of the application.
* It is a part of React and doesn’t require any additional libraries its Included within React, making it easy to get started
* Context API is relatively easy to set up for small to medium scale applications. It is a great choice when you need to pass data through multiple layers of components without prop drilling.
* Compared to Redux, Context API involves less boilderplate code.

You don’t need to define actions,reducers and stores.

**Use Redux:**

* For complex global state management.
* When you need predictable state changes with strict patterns (actions, reducers).
* In larger applications with a significant amount of shared state.
* You get advantage of Redux dev tools .

In some cases, a combination of both Context API and Redux may be used. Context API can be employed for simpler local state management, while Redux can handle more complex global state requirements in specific parts of the application. Ultimately, the choice depends on the specific needs and scale of your React application.

Higher Order Component (HOC)

A Higher Order Component (HOC) in React is a pattern that involves a function that takes a component as an argument and returns a new enhanced component. The purpose of an HOC is to reuse component logic, provide additional props, or alter the behavior of a component in a modular and reusable way.

import React from 'react'

import Home from './Home'

function App(props) {

  return (

    <>

    <h1> Iron man with = {props.costume} </h1>

    </>

  )

}

export default Home(App)

function Home(Component) {

  return function NewComponent(){

      return <>

      <div style={{ background : "green" }} >

      <Component costume = "Iron Suit"  />

      </div>

      </>

  }

}

Redux

Redux is a state management library commonly used with React to manage the state of an application in a predictable and centralized way. It helps to manage the state of your application in a single global store, making it easier to understand, debug, and maintain.

createStore

**createStore** is a function provided by the Redux library that is used to create a Redux store. In the context of Redux, a store is an object that holds the complete state of your application. The store is the single source of truth, and it allows you to manage the state in a predictable and centralized manner.

import { createStore } from 'redux';

import Reducer from './reducer'; // Assuming you have a root reducer

const store = createStore(Reducer,window.\_\_REDUX\_DEVTOOLS\_EXTENSION\_\_ && window.\_\_REDUX\_DEVTOOLS\_EXTENSION\_\_())

export default store;

Provider

**Provider** is a component provided by the **react-redux** library, and it plays a crucial role in integrating Redux with React applications. It is used to wrap your entire React application and make the Redux store accessible to all components in the component tree without having to pass it explicitly through each level.

import React from 'react';

import ReactDOM from 'react-dom';

import { Provider } from 'react-redux';

import store from './store'; // Assuming you have a Redux store

import App from './App'; // Your main application component

ReactDOM.render(

  <Provider store={store}>

    <App />

  </Provider>,

  document.getElementById('root')

);

To set up **combineReducers** in a Redux application, you typically follow these steps:

1. **Create Individual Reducers:** Write separate reducer functions for different slices of your application state. Each reducer should manage a specific part of the state.

const obj = {

  Name : "a", // Name b

  Age : 20 // Age

export const nameReducer = (state = obj,action) => {

    return state.Name

export const ageReducer = (state = obj,action) => {

  return state.Age

1. **Combine Reducers:** Use the **combineReducers** function from the **redux** library to combine these individual reducers into a single reducer.

import { combineReducers } from 'redux';

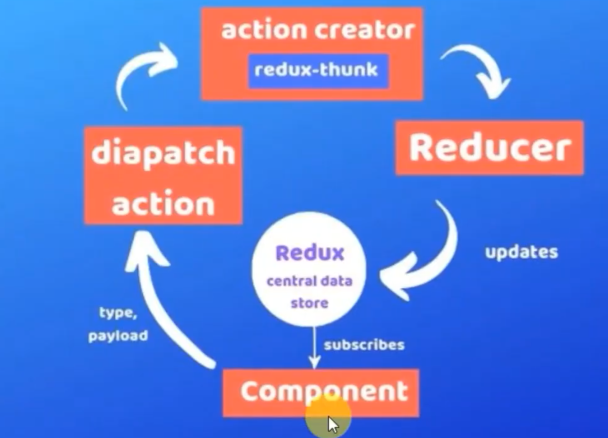
import { nameReducer,ageReducer } from './Reducer';

const rootReducer = combineReducers({ Name : nameReducer , Age : ageReducer  })

const store = createStore(rootReducer ,window.\_\_REDUX\_DEVTOOLS\_EXTENSION\_\_ && window.\_\_REDUX\_DEVTOOLS\_EXTENSION\_\_())

Redux Thunk

Redux Thunk is like a special helper that allows you to do more complex things, especially async tasks (like fetching data from an API), before updating your app's state.



Imagine you have a React-Redux app, and you want to fetch some data from an API and update your Redux store.

1. Without Redux Thunk:

const mapDispatchToProps = (dispatch) => {

  return {

    changeName : () => {

      console.log( "Called" )

      axios.get("https://jsonplaceholder.typicode.com/users")

      .then( (res) => {

        dispatch({ type : "change\_name" , payload : res.data[0].username   })

      } )

    }

  }

}

Redux Thunk is a middleware that allows you to dispatch functions (thunks) instead of plain action objects. These thunks can contain asynchronous logic and are executed by the Redux middleware.

Redux Thunk and similar middleware solutions provide a way to handle asynchronous operations in a more organized and standardized manner. They allow you to dispatch functions that can contain asynchronous logic and dispatch additional actions based on the asynchronous results. This helps separate concerns, improves code organization, and makes it easier to reason about your application's state and behavior.

1. With Redux Thunk:

Here's a simple example of how you can set up Redux Thunk in a React-Redux application:

import { createStore,applyMiddleware, compose } from 'redux';

import reportWebVitals from './reportWebVitals';

import {Reducer} from "./Reducer"

import { Provider } from 'react-redux';

import {thunk} from 'redux-thunk'

const composeEnhancers = window.\_\_REDUX\_DEVTOOLS\_EXTENSION\_COMPOSE\_\_ || compose;

const store = createStore(Reducer,composeEnhancers(applyMiddleware(thunk)))

**applyMiddleware** is used to add middleware (in this case, Redux Thunk) to the store. Middleware provides a way to interact with actions before they reach the reducer, enabling additional functionalities such as handling asynchronous actions.

function fetchName(){

  return (dispatch) => {

    axios.get("https://jsonplaceholder.typicode.com/users")

    .then( (res) => {

      dispatch({ type : "change\_name" , payload : res.data[0].username   })

    } )

  }

}

const mapDispatchToProps = (dispatch) => {

  return {

    changeName : () => {

        dispatch( fetchName() )

    }

  }

}

**Thunk**:

A thunk is a function that delays an action.

In the context of Redux Thunk, it's a special kind of action creator that returns a function instead of a plain action object.

**Why Thunk?:**

Thunks are useful for dealing with asynchronous operations, like fetching data from an API.

They let you do things (like API calls) before dispatching the actual action.

**Example:**

Imagine you want to fetch data from an API and update your Redux store.

With Redux Thunk, your action creator is a function that can perform async operations.

Without Redux Thunk, actions are simple objects, and handling async tasks becomes tricky.

With Redux Thunk, your action creator is a function that gets access to dispatch and can perform async tasks before dispatching the actual action.

This way, Redux Thunk makes it easier to manage asynchronous operations like API calls in your Redux app.

In a nutshell, Redux Thunk allows you to handle asynchronous tasks in a more organized way within your Redux actions. It adds a layer that lets you delay actions, making it easy to manage async operations in your Redux store.

# Redux Toolkit

The **Redux Toolkit** package is intended to be the standard way to write [Redux](https://redux.js.org/) logic. It was originally created to help address three common concerns about Redux:

* "Configuring a Redux store is too complicated"
* "I have to add a lot of packages to get Redux to do anything useful"
* "Redux requires too much boilerplate code"

Redux Toolkit also includes a powerful data fetching and caching capability that we've dubbed ["RTK Query"](https://redux-toolkit.js.org/introduction/getting-started#rtk-query). It's included in the package as a separate set of entry points. It's optional, but can eliminate the need to hand-write data fetching logic yourself.

By using Redux Toolkit, developers can write Redux logic with less boilerplate code, which makes the codebase more maintainable and easier to understand. It is designed to be opinionated, providing a set of best practices and conventions for structuring Redux code, while still allowing developers the flexibility to customize and extend as needed. Overall, Redux Toolkit is aimed at improving the developer experience when working with Redux in React applications.

**configureStore**

n Redux Toolkit, the configureStore function is a utility provided to simplify the process of creating a Redux store. It encapsulates several common configurations and best practices, making it easier for developers to set up a Redux store with minimal boilerplate code.

Here's a basic example of using configureStore:

import { configureStore } from '@reduxjs/toolkit';

import rootReducer from './reducers'; // Assuming you have a root reducer

const store = configureStore({

  reducer: rootReducer,

  // Other optional configurations can be added here

});

export default store;

**createReducer**

createReducer function from the @reduxjs/toolkit package, which is a part of Redux Toolkit. This function simplifies the process of creating Redux reducers by allowing you to define a reducer as a mapping from action types to functions, using the createSlice function.

import { createReducer } from "@reduxjs/toolkit"

let initialState = {

    Name : "A",

    Age : 20

}

export default  createReducer(initialState , (builder) => {

    builder.addCase( "update\_name" , (state,action) => {

        state.Name = action.payload

    } )

})

In this example, createReducer is used to define a r. The builder object passed to the callback provides methods like **addCase** to associate action types with corresponding reducer functions.

This code assumes that you have actions with types “update\_name” that you dispatch to trigger the corresponding reducer functions. Make sure to replace these action types with the actual types used in your application.

**useSelector**

**useSelector** hook from the react-redux library, which is a hook that allows you to extract data from the Redux store in a React component.

import React from 'react'

import { useSelector } from 'react-redux'

function App() {

 let {Name,Age} = useSelector( (state) => {

    return state

  })

  return (

    <>

    <h2>App Component</h2>

    <p> Name : {Name} </p>

    </>

  )

}

export default App

useSelector hook is used to access the property from the Redux store. The provided callback function takes the entire Redux state as an argument, and you can return the specific piece of state that your component needs.

In development, a check is conducted on the result returned by the selector. It warns in the console if the result is the same as the parameter passed in, i.e. the root state.

A useSelector call returning the entire root state is almost always a mistake, as it means the component will rerender whenever anything in state changes. Selectors should be as granular as possible, like state => state.some.nested.field.

  // BAD: this selector returns the entire state, meaning that the component will rerender unnecessarily

const { count, user } = useSelector((state) => state)

// GOOD: instead, select only the state you need, calling useSelector as many times as needed

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

const user = useSelector((state) => state.auth.currentUser)

**useDispatch**

the **useDispatch** hook from the react-redux library. This hook allows you to get access to the dispatch function, which is used to dispatch actions to the Redux store.

import React from 'react'

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

function App() {

 let Name = useSelector( (state) => {

    return state.Name

  })

  let dispatch = useDispatch();

  function UpdateName(){

    dispatch({ type : "update\_name" , payload : "B" })

  }

  return (

    <>

    <h2>App Component</h2>

    <p> Name : {Name} </p>

    <button onClick={ () => { UpdateName() } } > Update Name </button>

    </>

  )

}

export default App

Action creator

In Redux, an action creator is a function that creates and returns an action object. Actions are plain JavaScript objects that describe changes to the state of the application. Action creators serve as a way to encapsulate the logic of creating these action objects, making it easier to manage and dispatch them throughout the application.

import {updateNameAction} from "./Actions"

  let dispatch = useDispatch();

  function UpdateName(){

    dispatch(updateNameAction())

  }

Actions.js

export const updateNameAction = () => {

    return { type : "update\_name" , payload : "B" }

}

**import { createAction } from "@reduxjs/toolkit"**

The **createAction** function is a utility provided by the @reduxjs/toolkit library, which is part of Redux Toolkit. This function simplifies the process of creating action creators in a more concise way compared to the traditional Redux approach.

import {updateNameAction} from "./Actions"

  function UpdateName(){

    dispatch(updateNameAction("B"))

  }

import { createAction } from "@reduxjs/toolkit"

export const updateNameAction = createAction("update\_name")

In this example, createAction takes a single argument, which is the type of the action. It returns an action creator function. When you invoke this function, it generates an action object with the specified type. If the action creator has a payload, you can pass the payload as an argument when calling the function.

**Thunk in Reduxjs/toolkit**

Redux Toolkit includes the redux-thunk middleware by default. When you create a Redux store using configureStore from @reduxjs/toolkit, it automatically includes redux-thunk as part of the middleware setup. This makes it easy to use thunks in your Redux application without needing to manually add the middleware.

 import {updateNameAction} from "./Actions"

  function UpdateName(){

    dispatch(updateNameAction())

  }

import axios from "axios"

export const updateNameAction = () => {

    return async (dispatch) => {

      let res =   await axios.get("https://jsonplaceholder.typicode.com/users")

      dispatch({ type : "update\_name" , payload : res.data[0].name })

    }

}

With Redux Toolkit, you get the benefits of the redux-thunk middleware without the need to manually include it in your middleware setup. This helps streamline the development process and keeps the codebase clean.

createSlice in redux

**createSlice** is a utility function provided by the @reduxjs/toolkit library, which is a part of Redux Toolkit. It is used to create a slice of the Redux store, including the reducer function and the associated action creators. This utility simplifies the process of defining reducers and action creators by combining them into a single, compact structure.

Creating a slice

import { createSlice } from "@reduxjs/toolkit";

const initialState = {

    Name : "A",

    Age : 20

}

const userReducer = createSlice({

    name : "user",

    initialState,

    reducers : {

        changeName : (state,action) => {

            console.log("Calledd...")

            state.Name =  action.payload

        },

        changeAge : (state,action) => {

            state.Age =  action.payload

        }

    }

})

export const { changeName, changeAge } = userReducer.actions;

export default userReducer.reducer;

setting up data inside redux

import { configureStore } from '@reduxjs/toolkit';

import { Provider } from 'react-redux';

import userReducer from "./Reducer"

let store = configureStore({

   reducer : {

    user : userReducer

   }

 })

Using and updating data in Redux

import React from 'react'

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

import { changeName ,changeAge } from "./Reducer"

function App() {

   let {Name,Age} = useSelector( (state) => {

    return state.user

  } );

  let dispatch = useDispatch();

  const changeNameFn = (name) => {

    dispatch(changeName(name))

  }

  const changeAgeFn = (Age) => {

    dispatch(changeAge(Age))

  }

  return (

    <>

    <div>App</div>

    <p> Name : {Name} </p>

    <button onClick={ () => { changeNameFn("B")  } } > change Name </button>

    <p> Age : {Age} </p>

    <button onClick={ () => {  changeAgeFn(30) } } > change Age </button>

    </>

  )

}

export default App

createAsyncThunk

**createAsyncThunk** is a utility function provided by the Redux Toolkit, a set of tools and guidelines for building Redux applications in a more efficient and ergonomic way. It simplifies the process of handling asynchronous logic, such as making API calls, within Redux actions.

To use **createAsyncThunk**, you typically define a thunk function that dispatches three different actions based on the different states of the asynchronous operation: **pending**, **fulfilled**, and **rejected**. Here's an example of how you might use createAsyncThunk:

In this example:

**createAsyncThunk** is used to create the **fetchUser** thunk.

Three actions are dispatched automatically: **fetchUser.pending**, **fetchUser.fulfilled**, and **fetchUser.rejected**. These actions represent the different states of the asynchronous operation.

The **extraReducers** section of **createSlice** handles these actions to update the Redux state accordingly.

By using **createAsyncThunk**, you can manage asynchronous operations in a more organized and standardized way, reducing boilerplate code in your Redux slices.

Top of Form

import { createSlice , createAsyncThunk } from "@reduxjs/toolkit";

import axios from 'axios';

export const fetchUserOnline = createAsyncThunk(

    'Get Users',

    async () => {

        let res = await axios.get("https://jsonplaceholder.typicode.com/users")

        return res.data[0].name

    }

  )

const initialState = {

    Name : "A",

    Age : 20

}

const userReducer = createSlice({

    name : "user",

    initialState,

    reducers : {

        changeName : (state,action) => {

            console.log("Calledd...")

            state.Name =  action.payload

        },

        changeAge : (state,action) => {

            state.Age =  action.payload

        }

    },

    extraReducers : (builder) => {

        builder.addCase(fetchUserOnline.fulfilled, (state, action) => {

            // Add user to the state array

            console.log(state);

            state.Name = action.payload

          })

          builder.addCase(fetchUserOnline.pending, (state, action) => {

            // Add user to the state array

            state.Name = "Loading...."

          })

    }

})

useRef()

**useRef** is used in react for several purposes,primary related to managing and interacting with dom elements. **useRef** is a React Hook that provides a way to create mutable object references that persist across renders without causing re-renders when the reference changes. It is primarily used for accessing and interacting with the underlying DOM elements or for persisting values across renders without triggering a component re-render.

Showing no of times a component is rendered on Screen .

**Causing Infinite Loop.**

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

function App(){

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

const [renderCount,setRenderCount] = useState(0)

useEffect( () => {

setRenderCount(  (prevRenderCount) => prevRenderCount + 1)

})

return (

  <>

<input  value = {name} onChange={ (e) => {  setName(e.target.value) } }  />

<p> Name is {name} </p>

<p> I Rendered {renderCount} times   </p>

</>

)

}

export default App;

**Keeping render counts but component re-renders**

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

function App() {

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

  const [renderCount, setRenderCount] = useState(0)

  useEffect(() => {

    setRenderCount((prevRenderCount) => prevRenderCount + 1)

  }, [name])

  return (

    <>

      <input value={name}

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

      />

      <p> Name is {name} </p>

      <p> I Rendered {renderCount} times   </p>

    </>

  )

}

export default App;

**useRef Does not cause re-renders**

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

function App() {

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

  const renderCount = useRef(0)

  useEffect( ()  => {

    renderCount.current = renderCount.current + 1

    } )

  return (

    <>

      <input value={name}

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

      />

      <p> Name is {name} </p>

      <p> I Rendered {renderCount.current} times   </p>

    </>

  )

}

export default App;

**Selection using Dom vs useRef**

import React,{useRef} from 'react'

function App() {

  function btnClick(){

  const inputElement  =  document.getElementById("inp");

  inputElement.focus();

  }

  return (

    <>

    <h1> App Component  </h1>

    <input placeholder='Enter name' id = "inp" />

    <button onClick={ btnClick } >  click to focus </button>

    </>

  )

}

export default App

import React,{useRef} from 'react'

function App() {

  let inputElement = useRef();

  function btnClick(){

  inputElement.current.focus();

  }

  return (

    <>

    <h1> App Component  </h1>

    <input ref={inputElement} placeholder='Enter name' id = "inp" />

    <button onClick={ btnClick } >  click to focus </button>

    </>

  )

}

export default App

Changing background color (Dom Manipulation) using useRef

import React, { useRef } from 'react';

const MyComponent = () => {

  // Create a useRef to hold a reference to a DOM element

  const myElementRef = useRef(null);

  const changeColor = () => {

    if (myElementRef.current) {

      myElementRef.current.style.backgroundColor = 'lightblue';

    }

  }

  return (

    <div>

      {/\* Attach the ref to a specific DOM element \*/}

      <div ref={myElementRef}>This is my element</div>

      <button onClick={ () => {

        changeColor()

      } } > change color  </button>

    </div>

  );

};

export default MyComponent;

Advantages of using useRef instead of document object

1. **Reconcilation and virtual dom** - React uses a virtual dom and a reconciliation process to effectively update the actual dom . when you manipulate the dom using the ‘ducument’ object react may not be aware of the changes you’ve made leading to conflicts and issues.
2. **Re-rendering and state management** – React components re-render based on their state and props. When you use ‘useRef’ to interact with Dom elements ,it doesn’t trigger component re-render .This can be beneficial for managing the Dom without causing unnecessary re-renders of your component .

useMemo

The **useMemo** hook in React is useful for optimizing performance by avoiding unnecessary calculations or computations during renders. By using useMemo, you can avoid the recomputation of expensive operations on each render cycle, especially in scenarios where the computation involves heavy calculations or data fetching.

Without useMemo

import React,{useState} from 'react'

function App() {

  const [count,setCount] = useState(0)

  const [item,setItem] = useState(0)

  function DoubleCount(){

    console.log( "DoubleCount called" )

    return count \* 2

  }

  return (

    <>

    <h1>  App Component </h1>

    <p> Count : {count} </p>

    <p> Item : {item} </p>h

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

    <button onClick={ () => { setItem( item + 1 ) } } > Increate item </button>

     <h2> {DoubleCount()} </h2>

  </>

  )

}

export default App

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

function App() {

  const [count,setCount] = useState(0)

  const [item,setItem] = useState(0)

  const doubleCount =  useMemo( () => {

    console.log( "DoubleCount called" )

    return count \* 2

  },[count])

  return (

    <>

    <h1>  App Component </h1>

    <p> Count : {count} </p>

    <p> Item : {item} </p>

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

    <button onClick={ () => { setItem( item + 1 ) } } > Increate item </button>

     <h2> {doubleCount} </h2>

  </>

  )

}

export default App

useReducer

**useReducer** is a React hook that manages state in functional components. It's an alternative to useState and is particularly useful when the state logic is complex and involves multiple sub-values or when the next state depends on the previous one. It's similar to how Redux manages state in a Redux store.

The useReducer hook takes in a reducer function and an initial state. It returns an array containing the current state and a dispatch function. The dispatch function is used to trigger state transitions by specifying an action that describes the type of state change to perform.

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

Here, state represents the current state managed by useReducer, dispatch is a function used to dispatch actions, reducer is a function that specifies how the state should change based on the action dispatched, and initialState is the initial state value.

The reducer function takes two arguments: the current state and an action. It returns the new state based on the action type. The action is typically an object with a type property that describes what kind of state change should occur, along with any additional data needed for the update.

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:

return state;

}

}

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

Then, you can use dispatch to update the state by passing an action:

dispatch({ type: 'increment' });

dispatch({ type: 'decrement' });

This will trigger the reducer function, which will return the new state based on the type of action dispatched.

import React,{useReducer} from "react"

const App = () => {

const initialState = { count: 0 };

function reducer(state, action) {

  console.log(action);

  switch (action.type) {

    case 'increment':

      return { count: state.count + 1 };

    case 'decrement':

      return { count: state.count - 1 };

    default:

      return state;

  }

}

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

return (

  <>

  <h1> Counter </h1>

   <h3> Count = {state.count}  </h3>

  <br />

  <br />

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

  <br />

  <br />

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

  </>

)

}

export default App;

contextApi

In React, the Context API is a feature that allows you to share state between components without having to pass props through each level of the component tree. It provides a way to pass data through the component tree without having to pass props down manually at every level.

The Context API consists of two main parts:

1. **Provider:**

* The **Provider** component is used to wrap a portion of your component tree.
* It takes a **value** prop, which is the data you want to share.
* The data provided by the **Provider** is then accessible to any child components within its hierarchy.

import {createContext} from 'react'

export const GlobalContext = createContext();

function App() {

  const [data,setData] = useState({ a : 10 })

  return (

    <>

    <h1> App Component  </h1>

    <GlobalContext.Provider value = {{ a :  data.a , setA : () => { setData({ a : 20 }) } }} >

          <ChildComponent />

    </GlobalContext.Provider>

    </>

  )

}

export default App

1. **Consumer:**

* The **Consumer** component is used to access the data provided by the **Provider**.
* It can be used within a component to consume the context and render content based on that context.

import React from 'react'

import {GlobalContext} from "./App.js"

function Child2() {

  return (

    <>

    <GlobalContext.Consumer>

        { (data) => {

            console.log(data);

        } }

    </GlobalContext.Consumer>

    </>

  )

}

export default Child2

1. **useContext Hook:**

* The **useContext** hook is another way to consume the context in a functional component.
* import React,{useContext} from 'react'

import { GlobalContext } from './App'

* function Child2() {
* let data = useContext(GlobalContext);
* return (
* <>
* <h1> Child2 Component </h1>
* </>
* )
* }

By using the Context API, you can avoid prop drilling (passing props through multiple layers of components) and make the state accessible to the components that need it without explicitly passing it down through each level of the component tree.

The **Context API** and **Redux** are both state management solutions in React, but they serve different purposes and have different use cases. Here are the key differences between Context API and Redux:

1. **Use Case and Complexity:**

* **Context API:** It is a built-in feature of React designed for managing the global state of a React application. It is ideal for simpler state management needs where you have a few pieces of shared state but don't want to pass props down through many layers of components.
* **Redux:** It is a state management library that provides a predictable state container. It is suitable for larger and more complex applications where the state management requirements are extensive. Redux introduces concepts like actions, reducers, and a unidirectional data flow.

1. **Scope:**

* **Context API:** Primarily designed for local state management within a component tree. While it can be used for global state, it may not scale well for very large applications due to potential performance issues.
* **Redux:** Specifically designed for managing global state. It allows you to maintain a centralized store that holds the entire state of your application, and components can connect to this store as needed.

While the Context API in React is a powerful tool for managing state and avoiding prop drilling, it's important to be aware of potential performance considerations, especially in scenarios involving deeply nested components or frequent updates to the context value. Here are some performance considerations associated with the Context API:

**Re-rendering of Consumers:**

* When the context value changes, all components consuming that context using the **Consumer** component or the **useContext** hook will re-render.
* In scenarios where the context provider updates frequently, this can lead to unnecessary re-renders of many components in the subtree, impacting performance.

**Component Granularity:**

* If the context provider is too high in the component tree, it can result in a large number of components re-rendering when the context value changes, even if some of those components don't actually depend on the context.
* Consider placing the context provider closer to the components that actually need the context.

1. **Number of Contexts:**

* If your application uses multiple contexts, each context update may trigger a re-render of components consuming that context.
* Be mindful of the number of contexts used and their update frequency.

It's essential to profile and measure the performance of your application using tools like React DevTools or other profiling tools to identify any bottlenecks and optimize accordingly. Context API is a valuable tool, and with careful design and consideration, its performance impact can be minimized.

To mitigate these performance concerns, consider the following strategies:

* **Memoization:**
  + Memoize context values using techniques like **useMemo** or memoized functions to avoid unnecessary re-renders.
* **Optimizing Providers:**
  + Optimize the context provider to avoid unnecessary updates. For example, memoize the provider component using **React.memo** or wrap it with **React.memo** if it doesn't depend on props or state changes.
* **Component Granularity:**
  + Place context providers closer to the components that actually depend on the context to minimize the number of components affected by context updates.
* **Selective Context Updates:**
  + If possible, design your context providers to update only when necessary, avoiding unnecessary re-renders.

Custom Hook

Creating custom React hooks is a powerful way to encapsulate and reuse logic in your components. A custom hook is a JavaScript function whose name starts with "use" and may call other hooks

Using the custom hook in App.js

import React from 'react';

import useCustomHook from './CustomHook';

const App = () => {

  const { value, handleChange,addItem } = useCustomHook('');

  return (

    <div>

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

      <p>Current Value: {value}</p>

      <button onClick={ () => { addItem() } } > store value </button>

    </div>

  );

};

export default App;

Creating custom hook In a separate file

import React,{useState} from "react"

const useCustomHook = (initialValue) => {

    const [value, setValue] = useState(initialValue);

    const handleChange = (newValue) => {

      setValue(newValue);

    };

    const addItem = () => {

       localStorage.setItem("A" , value)

    }

    return {

      value,

      handleChange,

      addItem

    };

  };

  export default useCustomHook

Flux

Flux is an architectural pattern used in building user interfaces, particularly in the context of React applications. It was developed by Facebook to address the challenges of managing state in large and complex applications. Flux complements React's component-based architecture by providing a unidirectional data flow and a clear separation of concerns.

Here's a simplified flow of how React and Flux work together:

1. **User Interaction:**
   * A user interacts with a React component, such as clicking a button.
2. **Action Dispatch:**
   * The React component dispatches an action, describing the user interaction.
3. **Dispatcher Handling:**
   * The Dispatcher receives the action and dispatches it to the relevant stores.
4. **Store Update:**
   * The stores respond to the action, update their state, and emit a change event.
5. **Component Re-render:**
   * React components listening to the changed store(s) receive the update and re-render with the new state.

This unidirectional flow helps manage the application state in a clear and maintainable way, making it easier to understand how data changes over time and reducing the risk of unexpected side effects. While the original Flux pattern was introduced by Facebook, various Flux-inspired libraries, such as Redux, have gained popularity within the React ecosystem.

What is testing

Testing involves the process of verifying that your code works as expected by creating automated tests. There are various types of tests such as unit tests, integration tests, and end-to-end tests, each focusing on different aspects of your codebase.

Testing is crucial for maintaining code quality, catching bugs early in the development process, and ensuring that changes to your codebase do not break existing functionality



* **Unit Tests**: These test small, isolated parts of your code, typically individual functions or modules, to ensure they work as intended



**Integration Tests**: These verify that different parts of your application work together correctly. They test the interactions between various modules, components, or services to ensure they integrate seamlessly.

****

**End-to-End Tests**: These simulate real user scenarios by testing the entire application workflow from start to finish

Let's create a simple example of testing a sum function using Jest in a React project. We'll create a file named **sum.js** containing a simple function that adds two numbers, and then we'll write a Jest test for that function.

Create a file named **sum.js**:

// sum.js

function sum(a, b) {

  return a + b;

}

export default sum;

Now, create a test file named **sum.test.js** in the same directory:

// sum.test.js

import sum from './sum';

// Test case: Check if the sum function adds two numbers correctly

test('adds 1 + 2 to equal 3', () => {

  expect(sum(1, 2)).toBe(3);

});

// Test case: Check if the sum function handles negative numbers correctly

test('adds -1 + 2 to equal 1', () => {

  expect(sum(-1, 2)).toBe(1);

});

// Test case: Check if the sum function handles floating-point numbers correctly

test('adds 0.1 + 0.2 to equal 0.3', () => {

  expect(sum(0.1, 0.2)).toBeCloseTo(0.3);

});

Checking if an element exists in Dom by ID

App.js

import React from 'react';

const MyComponent = () => {

  return (

    <div>

      <h1 data-testid ="main-heading">Hello, World!</h1>

      <p>This is a sample component.</p>

    </div>

  );

};

export default MyComponent;

app.test.js

import React from 'react';

import { render, screen } from '@testing-library/react';

import MyComponent from './Login';

test('selects element by id', () => {

  // Render the component

  render(<MyComponent />);

  // Select the element by its id

  const headingElement = screen.getByTestId("main-heading")

  // Assert that the element is in the document

  expect(headingElement).toBeInTheDocument();

  // Assert that the element has the correct text content

  expect(headingElement).toHaveTextContent('Hello, World!');

});

Tesing input Element

Login.js

import React from 'react'

function Login() {

  return (

    <div style={{ textAlign : "center" }} >

      <input type='text' placeholder='username' value = "helloo" />

      <br />

      <input type='password' placeholder='Enter your password' />

      <br />

      <button> Login </button>

      </div  >

  )

}

export default Login

App.test.js

import { render, screen } from '@testing-library/react';

import Login from "./Login"

test("Get input Element" , () => {

  render(<Login />);

  const userInputElement = screen.getByPlaceholderText(/username/i);

  expect(userInputElement.value).toBe("hello")

} )

Testing GetApi Call

import React from 'react';

import { render, screen } from '@testing-library/react';

import {fetchUserData} from './fetchUserData';

it('fetches user data successfully', async () => {

  // Mock the response from the API

  const mockUserData = {

    "id": 1,

    "name": "Leanne Graham",

    "username": "Bret",

    "email": "Sincere@april.biz",

    "address": {

    "street": "Kulas Light",

    "suite": "Apt. 556",

    "city": "Gwenborough",

    "zipcode": "92998-3874",

    "geo": {

    "lat": "-37.3159",

    "lng": "81.1496"

    }

    },

    "phone": "1-770-736-8031 x56442",

    "website": "hildegard.org",

    "company": {

    "name": "Romaguera-Crona",

    "catchPhrase": "Multi-layered client-server neural-net",

    "bs": "harness real-time e-markets"

    }

    };

  // Call the fetchUserData function

  const userData = await fetchUserData();

  // Assert that the function returns the expected data

  expect(userData).toEqual(mockUserData);

});

App.js

export const fetchUserData = async () => {

  try {

    const response = await fetch('https://jsonplaceholder.typicode.com/users/1');

    const data = await response.json();

    return data;

  } catch (error) {

    throw new Error('Error fetching user data');

  }

};

Server-Side Rendering (SSR)

Server-Side Rendering (SSR) is a technique used in web development where the server generates the initial HTML content for a web page, and then sends this rendered HTML to the client

Here are some key points about Server-Side Rendering:

* 1. **Initial Content Generation**: When a user requests a page, the server executes the application code and generates the HTML for the requested page on the server itself. This HTML contains the initial state of the page.
  2. **Faster Initial Page Load**: With SSR, users receive a fully rendered page from the server, reducing the time it takes for the initial page to load. This can be beneficial for performance and user experience, especially on slower network connections or less powerful devices.
  3. **SEO Benefits**: Search engines typically index the content of web pages based on their initial HTML. SSR can improve SEO (Search Engine Optimization) because search engines can more easily index the content since it's present in the initial HTML.
  4. **Improved Perceived Performance**: Users see content faster since the initial HTML is delivered by the server. Any client-side JavaScript can then take over and enhance the interactivity, but the user doesn't have to wait for JavaScript to load before seeing something on the page.
  5. **Consistent Rendering:** SSR helps ensure a consistent initial rendering across different devices and browsers since the server generates the HTML.

Create Simple App with npx create-react-app app-name

Change the code in index.js file

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

replace with

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

Install Express with below command

npm install express

Install babel,babel-react and ignore-style with below command

npm install @babel/register @babel/preset-env @babel/preset-react ignore-styles

**Create a new folder called server, then go into it and create a file named server.js**

import path from 'path'

import fs from 'fs'

import express from 'express'

import React from 'react'

import ReactDOMServer from 'react-dom/server'

import App from '../src/App'

const PORT = 8080

const app = express()

const router = express.Router()

const serverRenderer = (req, res, next) => {

  fs.readFile(path.resolve('./build/index.html'), 'utf8', (err, data) => {

    if (err) {

      console.error(err)

      return res.status(500).send('An error occurred')

    }

    return res.send(

      data.replace(

        '<div id="root"></div>',

        `<div id="root">${ReactDOMServer.renderToString(<App />)}</div>`

      )

    )

  })

}

router.use('^/$', serverRenderer)

router.use(

  express.static(path.resolve(\_\_dirname, '..', 'build'), { maxAge: '30d' })

)

// tell the app to use the above rules

app.use(router)

// app.use(express.static('./build'))

app.listen(PORT, () => {

  console.log(`SSR running on port ${PORT}`)

})

**Let’s create an entry point in server/index.js:**

require('ignore-styles')

require('@babel/register')({

  ignore: [/(node\_modules)/],

  presets: ['@babel/preset-env', '@babel/preset-react']

})

require('./server')

**Make Build and run with node** :

npm run build

node server/index.js