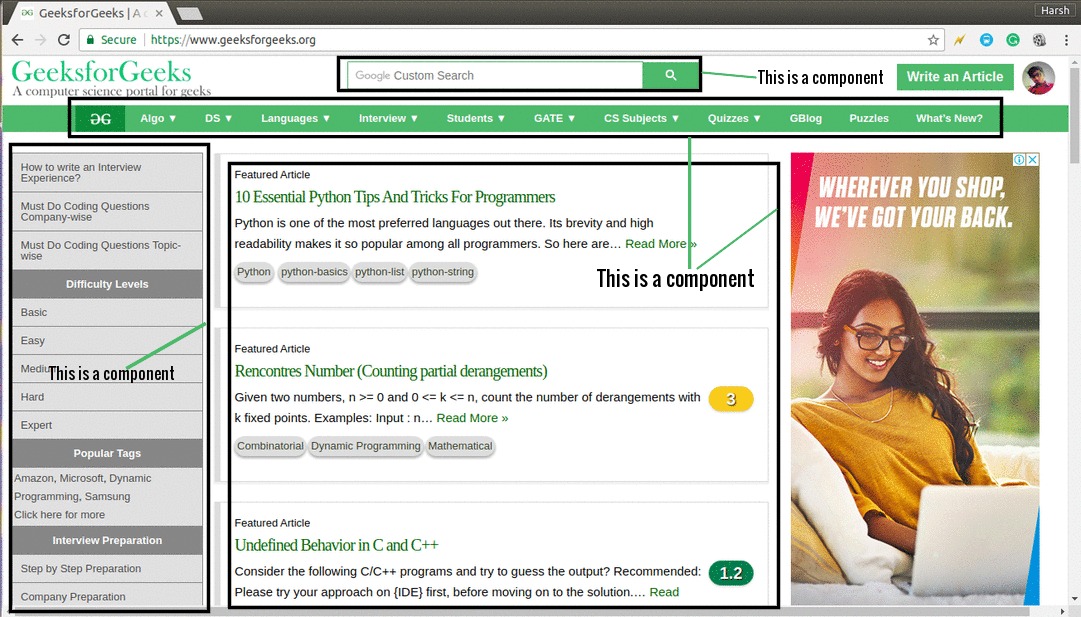
**Title : React components and state management in MERN Stack.**

**Introduction to React components :**

A Component is one of the core building blocks of React. In other words, we can say that every application you will develop in React will be made up of pieces called components. Components make the task of building UIs much easier. You can see a UI broken down into multiple individual pieces called components and work on them independently and merge them all in a parent component which will be your final UI.

You can see in the below image we have broken down the UI of GeeksforGeeks’s homepage into individual components.



Google’s custom search at the top can be seen as an individual component, the navigation bar can be seen as an individual component, the sidebar is an individual component, the list of articles or posts is also an individual component and finally, we can merge all of these individual components to make a parent component which will be the final UI for the homepage.

Components in React basically return a piece of JSX code that tells what should be rendered on the screen. In React, we mainly have two types of components:

[Functional Components](https://www.geeksforgeeks.org/reactjs-functional-components/): Functional components are simply javascript functions. We can create a functional component in React by writing a javascript function. These functions may or may not receive data as parameters, we will discuss this later in the tutorial. The below example shows a valid functional component in React:

[Class Components](https://www.geeksforgeeks.org/reactjs-class-based-components/): The class components are a little more complex than the functional components. The functional components are not aware of the other components in your program whereas the class components can work with each other. We can pass data from one class component to another class component. We can use JavaScript ES6 classes to create class-based components in React. The below example shows a valid class-based component in React:

class Democomponent extends React.Component {

render() {

return <h1>Welcome Message!</h1>;

}

}

The components we created in the above two examples are equivalent, and we also have stated the basic difference between a functional component and a class component. We will learn about more properties of class-based components in further tutorials. For now, keep in mind that we will use functional components only when we are sure that our component does not require interacting or work with any other component. That is, these components do not require data from other components however we can compose multiple functional components under a single functional component. We can also use class-based components for this purpose but it is not recommended as using class-based components without need will make your application in-efficient.

Rendering Components: In our previous post on [**rendering elements in React**](https://www.geeksforgeeks.org/reactjs-rendering-elements/) we have seen how elements initialized with DOM tags are rendered using ReactDOM.render() method. React is also capable of rendering user-defined components. To render a component in React we can initialize an element with a user-defined component and pass this element as the first parameter to ReactDOM.render() or directly pass the component as the first argument to the ReactDOM.render() method.

The below syntax shows how to initialize a component to an element:

const elementName = <ComponentName />;

In the above syntax, the *ComponentName* is the name of the user-defined component.

**Introduction to State management:**

React state management is a process for managing the data that React components need in order to render themselves. This data is typically stored in the component's state object. When the state object changes, the component will re-render itself. React state management is basically half of a React app.

# How to Manage State in React:

React state management is basically half of a React app. It includes all the data. The other half is the presentation including the HTML, CSS, and formatting. State and state management is relied on by presentation part of the app. The only time a React app will re-rendered is when state changes.

## The Four Kinds of React State to Manage:

When we talk about state in our applications, it’s important to be clear about what types of state actually matter.

There are four main types of state you need to properly manage in your React apps:

1. Local state
2. Global state
3. Server state
4. URL state

Local (UI) state – Local state is data we manage in one or another component.

Local state is most often managed in React using the useState hook.

For example, local state would be needed to show or hide a modal component or to track values for a form component, such as form submission, when the form is disabled and the values of a form’s inputs.

Global (UI) state – Global state is data we manage across multiple components.

Global state is necessary when we want to get and update data anywhere in our app, or in multiple components at least.

A common example of global state is authenticated user state. If a user is logged into our app, it is necessary to get and change their data throughout our application.

Sometimes state we think should be local might become global.

Server state – Data that comes from an external server that must be integrated with our UI state.

Server state is a simple concept, but can be hard to manage alongside all of our local and global UI state.

There are several pieces of state that must be managed every time you fetch or update data from an external server, including loading and error state.

Fortunately there are tools such as SWR and React Query that make managing server state much easier.

URL state – Data that exists on our URLs, including the pathname and query parameters.

URL state is often missing as a category of state, but it is an important one.  
In many cases, a lot of major parts of our application rely upon accessing URL state. Try to imagine building a blog without being able to fetch a post based off of its slug or id that is located in the URL!

**How to Manage Local State in React:**

Local state is perhaps the easiest kind of state to manage in React, considering there are so many tools built into the core React library for managing it.

useState is the first tool you should reach for to manage state in your components.

It can take accept any valid data value, including primitive and object values. Additionally, its setter function can be passed down to other components as a callback function (without needing optimizations like useCallback).

import { useState } from "react";

function Layout() {

const [isSidebarOpen, setSidebarOpen] = useState(false);

return (

<>

<Sidebar isSidebarOpen={isSidebarOpen} closeSidebar={() => setSidebarOpen(false)} />

{/\* ... \*/}

</>

);

}

useReducer is another option that can be used for either local or global state. It is similar in many ways to useState under the hood, although instead of just an initial state it accepts a reducer.

The benefit of useReducer is that it provides a built-in way to perform a number of different state operations with the help of the reducer function, which makes it more dynamic overall than useState.

You can see the benefit of useReducer versus useState in this vote tracking example. All we have to do to update state is pass the callback function dispatch a string (which is then passed to the reducer) instead of the new state itself

const initialState = { votes: 0 };

function reducer(state, action) {

switch (action.type) {

case 'upvote':

return {votes: state.votes + 1};

case 'downvote':

return {votes: state.votes - 1};

default:

throw new Error();

}

}

function VoteCounter() {

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

return (

<>

Current Votes: {state.votes}

<button onClick={() => dispatch({type: 'upvote'})}>Upvote</button>

<button onClick={() => dispatch({type: 'downvote'})}>Downvote</button>

</>

);

}

## How to Manage Global State in React:

Once you attempt to manage state across multiple components, things get a bit trickier.

You will reach a point in your application where patterns like “lifting state up” and passing callbacks down to update your state from components lead to lots and lots of props.

What do you do if you want to update a component’s state from basically anywhere in your app? You turn it into global state.

To manage it, however, you should opt for a third-party solution. Many developers are inclined to use built-in React features like the Context API to manage their state.

To be clear: the Context API is not a state management solution. It is a way to avoid problems like props drilling (creating a bunch of props in components that don’t need it), but it is only helpful for reading state, not updating it.

The reason to not use Context for global state management lies in the way it works. The default behavior for Context is to re-render all children components if the value provided to it as a prop changes.

For example, it is bad practice to combine useReducer and useContext:

function App() {

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

return (

<StateProvider.Provider value={{ state, dispatch }}>

<ComponentA />

<ComponentB />

<ComponentC />

</StateProvider.Provider>

)

}

In many cases, you do not want all children to update in response to a global state update, because all children may not be consuming or relying upon that global state. You only want to re-render if their props or state changes.

To manage your global state, reach for tried and tested third-party libraries like Zustand, Jotai, and Recoil.

## How to Manage Server State in React :

Server state can be deceptively challenging to manage.

At first, it seems you just need to fetch data and display it in the page. But then you need to display a loading spinner while you are waiting for the data. Then you need to handle errors and display them to the user as they arise.

What happens when there is a network error? Do I really need to hit my server every time my user visits the home page if the data hasn’t changed? Do I need to add useState and useEffect in every component I want to fetch my data?

To fix this, there are a couple of great libraries that make data fetching in React a breeze: SWR and React Query.

They not only give us a convenient hook to both get and change data from an API, but they keep track of all the necessary states and cache the data for us.

Here is an example of fetching a user’s profile from an API on the client. We call useSWR and specify the endpoint from which to request data, which is passed to our fetcher function and useSWR gives us both data and error state.

import useSWR from 'swr'

const fetcher = url => fetch(url).then(r => r.json())

function User() {

const { data, error } = useSWR('/api/user', fetcher)

if (error) return <div>failed to load</div>

if (!data) return <div>loading...</div>

return <div>hello {data.name}!</div>

}

SWR makes managing unsuccessful requests much easier and our components a lot nicer to look at.

Additionally, if you are performing the same operation over and over again, you use useSWR in your own custom hook to reuse across your app

function useUser (id) {

const { data, error } = useSWR(`/api/user/${id}`, fetcher)

return {

user: data,

isLoading: !error && !data,

isError: error

}

}

function Avatar ({ id }) {

const { user, isLoading, isError } = useUser(id)

if (isLoading) return <Spinner />

if (isError) return <Error />

return <img src={user.avatar} />

}

And finally, you can provide global options to useSWR, including your fetcher function (so you don’t need to pass it in every time) as well as a number of times to refetch data again after an error.

import useSWR, { SWRConfig } from 'swr'

function Admin () {

// no need to pass in the fetcher function

const { data: courses } = useSWR('/api/courses')

const { data: orders } = useSWR('/api/orders')

const { data: users } = useSWR('/api/users')

// ...

}

function App () {

return (

<SWRConfig

value={{

errorRetryCount: 2,

errorRetryInterval: 5000,

fetcher: (resource, init) => fetch(resource, init).then(res => res.json())

}}

>

<Admin />

</SWRConfig>

)

}

This is just a taste of the benefits of the SWR library, and React Query gives you just as many benefits, if not more.

Be sure to use either one for managing your server state. It will make your life so much easier.

## How to Manage URL State in React

To end a difficult topic on a positive note, URL state is largely already managed for you if you are using a framework like Next.js or the current version of React Router.

URL state is quite easy to manage, usually through custom hooks that give us all the information we need about our location, history, and pathname.

If you are using React Router, you can get all the information you need using useHistory or useLocation.

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

function BlogPost() {

const history = useHistory();

console.log("you are here: ", history.location);

const location = useLocation();

console.log('your pathname is: , location.pathname);

// ...

}

Additionally, if you have any route parameters that you need to use, for example to fetch data based off of, you can use the useParams hook.

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

function ChatRoom() {

const { roomId } = useParams();

const { chatRoom, isLoading, isError } = useChatRoom(roomId);

// ...

}

If you are using Next.js, almost everything can access directly from calling useRouter.

function Orders() {

const router = useRouter();

console.log('the entire url is: ', router.asPath);

console.log('your current route is: ', router.pathname);

console.log('your query params are: ', router.query);

function handleSubmit(item) {

setQuery("");

// push to new route

router.push(item.href);

closeDropdown();

}

// ...

}