

Real DOM vs Virtual DOM

React JS works on principle of VDOM

If any component updates, another copy of VDOM is created and is compared with the prev copy, uses diffing algo to track changes and updates only those changes in actual real DOM.

**JSX**

JSX – Javascript XML, JS extension, combination of JS and html in which we can merge JS and html easily

const App = () => {

  const user = "Durvesh";

  const age = 22;

*// function can also be defined here*

  return (

    <div>Hello, {user} {age} </div>

  )

}

**Return statement:**

 **Defines the UI:** The return statement determines what is displayed to the user.

 **Returns React Elements:** Typically, ***it returns JSX (JavaScript XML) or React elements*** that describe the desired UI.

 **Single Root Element:** The ***returned JSX must have a single parent element.*** This can be a single HTML tag or a React Fragment.

**useState hook – special types of functions that provide some powerful features, provides state management**

***useState is a React Hook that allows you to add state to your functional components.***

***State is a way to store and manage dynamic data in your components that can change over time, such as user inputs, form data, or any other data that affects what gets rendered on the screen.***

**Reactivity: When state changes, React automatically re-renders the component to reflect those changes in the UI.**

why not only function to change variable value?

Using only function you cant change the value of any variable, need to use useState hooks

How useState Works with the Virtual DOM

1. State Management:
   * When you use useState, you're creating a piece of state that holds dynamic data for a component. This state can change over time (e.g., through user interactions or API responses).
2. Triggering Updates:
   * Whenever you call the setter function (like setUser or setNum), you are signaling to React that something has changed.
   * For example, when you click the button to change the user name, setUser("Javed") is invoked, updating the user state.
3. Re-rendering the Component:
   * React then re-evaluates the component function to get the updated state.
   * It creates a new VDOM representation of that component based on the updated state.
4. VDOM Comparison:
   * React compares the new VDOM with the previous VDOM (the one that was created before the state change) to identify what has changed.
   * This process is called reconciliation. React determines the minimum number of updates needed to synchronize the real DOM with the updated VDOM.
5. Efficient Updates:
   * Only the parts of the DOM that need to be updated are changed, rather than re-rendering the entire DOM.
   * This is much more efficient and improves performance, especially in larger applications with many components.

**Counter in React**

import React, { useState } from "react";

const App = () => {

*// const user = "Durvesh";*

*// const age = 22;*

*// function can also be defined here*

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

  const[num, setNum] = useState(0);

  const changeUserName = () => {

    setUser("Javed")

  }

  return (

    <div>

    <h1> Hello, {user} </h1>

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

    <h2>{num}</h2>

    <button onClick={() => {setNum(num + 1)}}>Inc</button>

    <button onClick={() => {setNum(num - 1)}}>Dec</button>

    </div>

  )

}

export default App

**Form**

import React, { useState } from 'react';

const MyForm = () => {

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

const handleSubmit = (event) => {

event.preventDefault();

alert(`Submitted Name: ${name}`);

};

return (

<form onSubmit={handleSubmit}>

<label>

Name:

<input

type="text"

value={name}

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

/>

</label>

<button type="submit">Submit</button>

</form>

);

};

export default MyForm;

**In React, the *onSubmit event handler* for forms can indeed accept a callback function. This function is called when the form is submitted. Typically, you will want to handle form submission with this callback to prevent the default behavior (which is to refresh the page) and to perform any desired actions, like validating input or sending data to an API.**

**\*two way binding in React**

 Two-way data binding in React is achieved using controlled components and state management.

 The input value is controlled by the component's state, and changes in the input update the state, while changes in the state update the input.

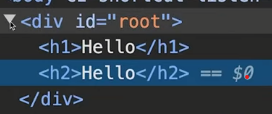
 This approach makes it easier to handle user input, validate forms, and manage dynamic data in your applications.

**Components**

Fragments: empty tags <> </>

Empty tags are not visible in browser

**React Fragments** are a powerful feature in React that help you manage your component's structure without introducing unnecessary elements into the DOM.



\*\***React Fragments** allow you to group a list of children elements without adding extra nodes to the DOM. Essentially, they let you return multiple elements from a component's render method without wrapping them in an additional HTML element like a <div>.

**React Fragments** are **not added to the DOM tree**.

***In order to make things reusable, we use components***

Eg: Nav bar, display card

import Navbar from './components/Navbar';

const App = () => {

  return (

    <>

      <Navbar />

    </>

  )

}

export default App

**Navbar.jsx**

import React, { useState } from "react";

const Navbar = () => {

  return(

    <>

    <nav className="flex bg-emerald-950 px-10 py-5 items-center justify-between">

      <h2 className="text-2xl">Durvesh</h2>

      <div className="flex gap-10">

        <h4 className="text-xl">About</h4>

        <h4 className="text-xl">Contact</h4>

        <h4 className="text-xl">Projects</h4>

        <h4 className="text-xl">Education</h4>

      </div>

    </nav>

    </>

  )

}

export default Navbar

**Props and Props Drilling – props are properities**

const App = () => {

  const username = "DJ";

  return (

    <>

      <Navbar />

      <div className='p-6 '>

        <Card user={username} />

      </div>

    </>

  )

}

Above, username is passed as prop named ‘user’ to Card component.

**Card.jsx**

**Card component receives prop as a Object, in h2, particular att. From object is accessed.**

const Card = (*props*) => {

    return(

        <div className="bg-white text-black inline-block p-6 text-center rounded">

            <h1>This is Card.</h1>

            <h2>{props.user}'s Favorites</h2>

        </div>

    )

}

export default Card;

**Rendering JSON data and passing JSON data as props to component**

**App.jsx**

return (

    <>

      <Navbar />

      <div className='p-6 '>

        {*/\* <Card user={username} /> \*/*}

        {users.map((*obj*, *idx*) => {

          return <Card key={idx} username={obj.name} age={obj.age} city={obj.city} photo={obj.profile\_photo} prof={obj.profession}/>

        })}

      </div>

    </>

  )

**Card.jsx**

const Card = (*props*) => {

    return(

        <div className="mr-7 bg-white text-black inline-block p-6 text-center rounded">

            <img className="ml-8 h-32 w-32 rounded-full mb-3" src="{props.photo}" alt="" />

            <h1 className="text-2xl font-semibold mb-4">{props.username}</h1>

            <h4 className="text-blue-400">{props.prof}</h4>

            <h2>{props.city}{props.age}</h2>

            <button className="mt-5 bg-emerald-700 text-white px-4 py-2 font-medium">View</button>

            {*/\* <h2>{props.user}'s Favorites</h2> \*/*}

        </div>

    )

}

export default Card;

-------------------------------------------------------------------------

**Data always flows from top to bottom**

**\*passing data from parent to child component**

**Props drilling:**

Prop drilling refers to the process of passing data from a parent component to a deeply nested child component through multiple intermediary components that do not need the data themselves. Essentially, props are "drilled" down through the component tree to reach the desired component.

**Integrating API (axios)**

What is axios?

**Axios** is a **Promise-based HTTP client** for both the browser and Node.js. It allows you to perform various HTTP requests (GET, POST, PUT, DELETE, etc.) to interact with APIs or other web services.

**Key Features of Axios:**

* **Promise-Based:** Simplifies asynchronous code with .then() and .catch() methods, and supports async/await syntax.
* **Interceptors:** Allows you to intercept requests or responses before they are handled by then or catch.
* **Automatic JSON Data Transformation:** Automatically transforms JSON data, making it easier to work with APIs.
* **Request and Response Transformation:** Modify requests and responses before they are handled.
* **Cancel Requests:** Supports request cancellation.
* **Timeouts:** Set timeouts for requests.
* **Protection Against XSRF:** Provides built-in protection against cross-site request forgery (XSRF).

In React applications, managing data from external sources is a common requirement. Axios simplifies this process by providing an easy-to-use API for making HTTP requests

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

  const getData = async () => {

    let response = await axios.get(

      "https://picsum.photos/v2/list?page=2&limit=10"

    );

    console.log(response.data);

    setData(response.data);

  };

**useEffect hook**

**now, I want data to be automatically displayed instead of button click, so how to do it?**

**useEffect hook – works on side stack**

The useEffect hook is one of the most powerful and essential features in React, especially for managing side

effects in functional components.

useEffect is a **React Hook** that allows you to perform **side effects** in your functional components. Side effects are operations that can affect other components or cannot be done during rendering, such as:

* **Data fetching:** Retrieving data from APIs.
* **Subscriptions:** Setting up subscriptions like WebSocket connections.
* **Manual DOM Manipulation:** Directly changing the DOM outside of React's rendering.
* **Timers:** Setting up intervals or timeouts.
* **Logging:** Logging information for debugging.

**Key Point:** Before Hooks, side effects were handled in class components using lifecycle methods like ***componentDidMount, componentDidUpdate, and componentWillUnmount***. useEffect brings this capability to functional components.

In React, components are meant to be **pure**—they should render UI based solely on props and state without causing side effects. However, real-world applications often require operations that go beyond rendering, such as fetching data or interacting with browser APIs. This is where useEffect comes into play:

* **Separation of Concerns:** Keeps your rendering logic separate from side-effect logic.
* **Control Over Side Effects:** Allows you to specify when and how side effects should run.
* **Cleanup:** Provides a way to clean up side effects to prevent memory leaks.

The useEffect hook accepts two arguments:

1. **Effect Function:** A function containing the side-effect logic.
2. **Dependency Array (Optional):** An array of dependencies that determine when the effect should run.

import React, { useEffect } from 'react';

function MyComponent() {

useEffect(() => {

// Side-effect logic here

return () => {

// Cleanup logic here (optional)

};

}, [/\* dependencies \*/]);

return (

<div>

{/\* Component JSX \*/}

</div>

);

}

**Understanding the Dependency Array**

The dependency array is crucial for optimizing when your side effects run. It tells React to only re-run the effect if one of the dependencies has changed.

**Scenarios:**

1. **No Dependency Array:**
   * **Behavior:** Runs after every render.
   * **Use Case:** Rarely used; generally not recommended due to performance implications.

useEffect(() => {

console.log('Effect runs after every render');

});

1. **Empty Dependency Array ([]):**
   * **Behavior:** Runs only once after the initial render (similar to componentDidMount).
   * **Use Case:** Fetching data on component mount.

useEffect(() => {

console.log('Effect runs once on mount');

}, []);

1. **With Dependencies ([dep1, dep2]):**
   * **Behavior:** Runs after the initial render and whenever any of the dependencies change.
   * **Use Case:** Updating the document title based on state changes.

useEffect(() => {

console.log('Effect runs on mount and when dep1 or dep2 changes');

}, [dep1, dep2]);

**Example:**

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

function Counter() {

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

// Effect runs only once on mount

useEffect(() => {

console.log('Component mounted');

}, []);

// Effect runs whenever 'count' changes

useEffect(() => {

console.log(`Count changed to ${count}`);

}, [count]);

return (

<div>

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

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

</div>

);

}

*\*Study cleanup functions part*

| **Aspect** | **Functional Components with useEffect** | **Class Components** |
| --- | --- | --- |

|  |  |  |
| --- | --- | --- |
| **Mounting** | useEffect(() => { ... }, []) | componentDidMount |

|  |  |  |
| --- | --- | --- |
| **Updating** | useEffect(() => { ... }, [deps]) | componentDidUpdate |

|  |  |  |
| --- | --- | --- |
| **Unmounting/Cleanup** | Cleanup function in useEffect | componentWillUnmount |

To understand where useEffect operates in the context of JavaScript's event loop, it's essential to grasp a few concepts:

**1. JavaScript Event Loop Basics**

* **Call Stack:** Executes synchronous code.
* **Microtask Queue:** Handles promises (.then, async/await).
* **Macrotask Queue:** Manages tasks like setTimeout, setInterval, and I/O operations.

**2. Execution Flow with useEffect**

* **Render Phase:** Synchronous operations happen here.
* **Commit Phase:** DOM updates are flushed to the screen.
* **useEffect Execution:** Scheduled as a **macrotask**, running **after** the current call stack and microtasks are completed.

**3. Why Macrotask?**

* **Asynchronous Nature:** useEffect should not block the main thread, ensuring that the UI remains responsive.
* **Post-Render Execution:** Running useEffect after DOM updates prevents janky UI behavior.

**🧩 Putting It All Together: An Example**

Let's illustrate how useEffect integrates with React's lifecycle and the JavaScript event loop through a practical example.

**Scenario: Fetching Data After Component Mount**

**Objective:** Fetch user data from an API after the component has mounted without blocking the UI.

**Step-by-Step Execution**

1. **Component Mounts:**
   * React begins rendering the component.
2. **Render Phase:**
   * React processes the JSX and prepares the virtual DOM.
   * **No Side Effects:** Only pure rendering logic occurs here.
3. **Commit Phase:**
   * React updates the actual DOM with the rendered content.
   * **UI Update:** The user sees the initial UI (e.g., a loading spinner).
4. **useEffect Callback Queued:**
   * After the commit phase, the useEffect callback is scheduled as a **macrotask**.
   * **Asynchronous Execution:** It won't run until the current call stack and microtasks are cleared.
5. **useEffect Runs:**
   * The callback executes, fetching data from the API.
   * **State Update:** Upon receiving data, setState is called, triggering a re-render.
   * **UI Update:** The component re-renders with the fetched data.

**React Router DOM**

**React Router DOM** is a popular routing library for React applications. It allows you to define multiple routes in your application, enabling navigation between different components or pages *without reloading the entire page.* This is fundamental for building **Single Page Applications (SPAs)** where content updates dynamically without full page refreshes.

Different types of Routers

Here, we use BrowserRouter, see in main.js

**Navbar.jsx**

const Navbar = () => {

  return(

    <>

    <nav className="flex bg-emerald-950 px-10 py-5 items-center justify-between">

      <h2 className="text-2xl">Durvesh</h2>

      <div className="flex gap-10">

**<a href="/about">About</a>**

        <h4 className="text-xl">About</h4>

        <h4 className="text-xl">Contact</h4>

        <h4 className="text-xl">Projects</h4>

        <h4 className="text-xl">Education</h4>

      </div>

    </nav>

    </>

  )

}

export default Navbar

if we just use a tag, the page refreshes

***therefore, we need <Link/> from react router dom, prevents page reloading so it follows SPA***

**Navbar.jsx with Link tag from react router dom**

import { Link } from "react-router-dom"

const Navbar = () => {

  return(

    <>

    <nav className="flex bg-emerald-950 px-10 py-5 items-center justify-between">

      <h2 className="text-2xl">Durvesh</h2>

      <div className="flex gap-10 text-lg-underline">

**<Link to='/'>Home</Link>**

**<Link to='/about'>About</Link>**

**<Link to='/contact'>Contact</Link>**

      </div>

    </nav>

    </>

  )

}

export default Navbar

**Context API**