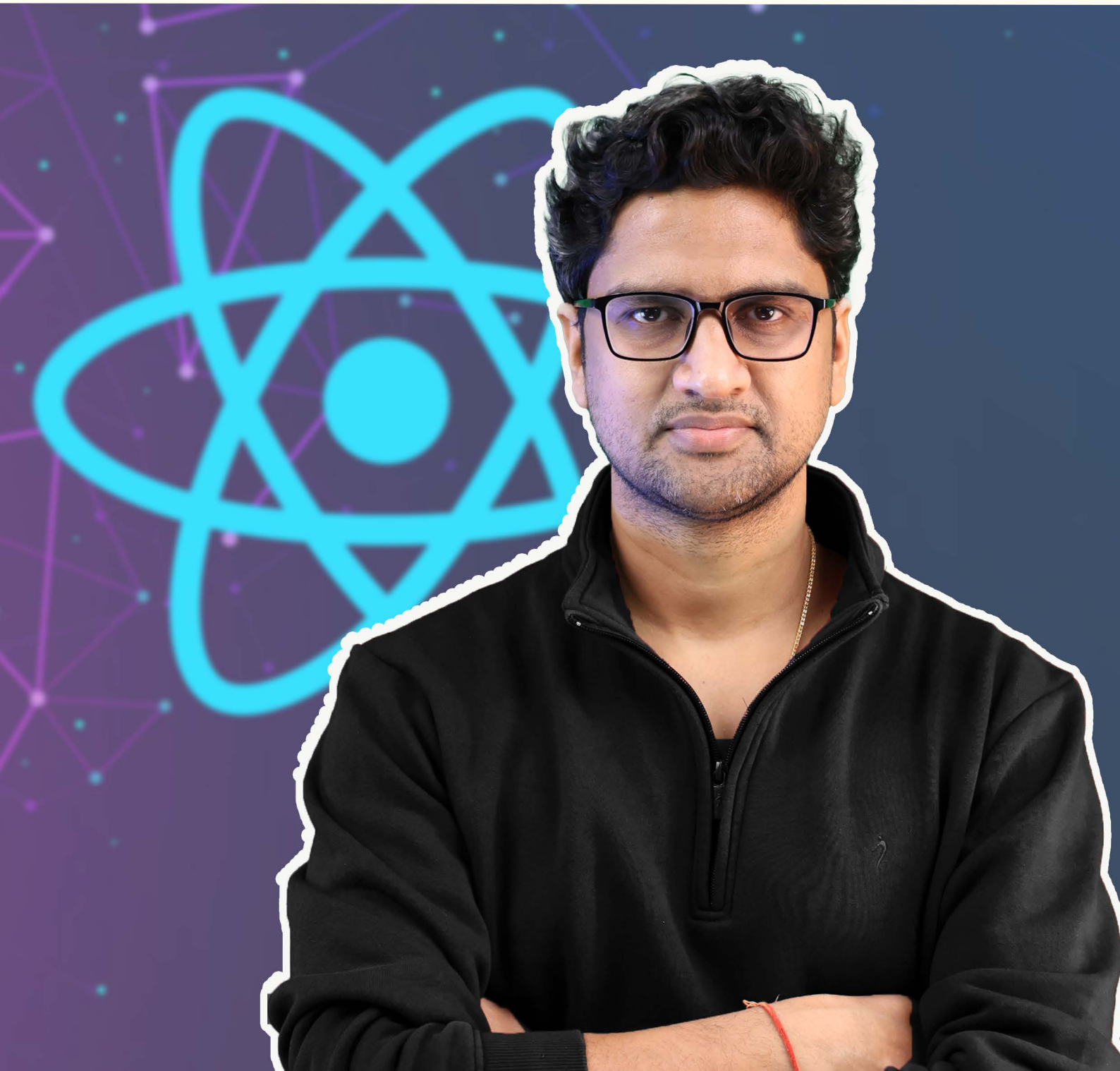
AIMERZ.ai
Aim.Act.Achieve

# Mastering React Hooks

**The Only Resource You Need to Master React Hooks from Start to Finish!**



AIMERZ.ai
Aim.Act.Achieve

# TABLE OF CONTENT

13 useRef

14 useContext

15 useSelector

16 useDispatch

17 useStore [ Zustand ]

18 useNavigate

19 useParams

20 useLocation

21 useSearchPar

22 useRoutes

23 useForm

24 useFormik

25 useDebounce

26 useThrottle

## Pre-Read Before You Dive In:

Congratulations on taking the first step toward mastering React Hooks! This eBook is designed to give you everything you need to deeply understand and confidently apply React Hooks in your projects. Here's how to make the most of it:

### 01 Understand the Concepts

Begin by reading the in-depth explanation of each hook. We've broken down the concepts into simple, easy-to-grasp sections, often accompanied by real-life analogies to make learning intuitive.

### 02 Take Your Time

Read through each concept carefully, and let the analogies and real-life examples help you grasp the core ideas. These examples are here to simplify complex topics and make them relatable.

### 03 Solve the Challenges

After learning each concept, put your knowledge to the test with a practical challenge. These challenges are designed to simulate real-world problems and help you solidify your understanding.

### 04 Review the Solutions

Once you've attempted the challenge, check out the detailed solution provided. Each solution comes with a step-by-step explanation to ensure you not only see the answer but also understand the reasoning and thought process behind it.

### 05 Reflect and Revise

Take a moment to reflect on the challenge and its solution. Revisit any concept that feels unclear, and think about how you can apply it to your own projects.

Every challenge you complete and every concept you understand brings you closer to mastering React Hooks. Celebrate your wins, big or small, and enjoy the journey!

**Now, it’s time to dive in and unlock the full potential of React Hooks. Let’s start learning, practicing, and growing together!**

**Let's start from the absolute scratch. Let's understand** why all developers need **React** and what happened before ?

## The Problem with the JavaScript DOM

The **DOM (Document Object Model)** is a way for JavaScript to interact with the elements on a webpage. But as websites and apps became more complex, working with the DOM directly had some challenges:

### Challenges of the DOM:

- **Too Slow for Complex Updates:**
  The DOM updates the entire page (or large parts of it) when something changes, even if only a small part needs to change. This slows down the app.
- **Hard to Manage State:**
  In large apps, keeping track of what data (state) should show where , gets complicated. A small mistake can cause bugs.
- **Messy Code:**
  Writing code to manually select elements, update their content, and keep everything in sync was repetitive and error-prone.
- **Re-rendering Issues:**
  Developers had to handle when and how to re-render elements manually, which added complexity.

Then react comes into picture to solve these problems

**React's Solution**

React was created by **Facebook** in 2011 to fix these problems. It introduced new ideas to make web development easier and faster.

**History of React**

- Around **2011**, Facebook's Ads app became hard to maintain because it had so many features and updates that caused problems (cascading updates).
- It became messy, hard to handle, and slowed the team down.

- **Jordan Walke**, a Facebook engineer, built a prototype called **FaxJS** (an early version of React) to fix this. It simplified updating parts of the app and made coding much easier.

## Key Events Timeline

1. **2013**: React was officially introduced by Facebook at a conference. At first, many developers didn't like it because it looked different from the tools they were used to. But Facebook started promoting it heavily.
2. **2014**: React gained popularity, especially with big companies like Netflix and Airbnb. They used it for building fast and interactive apps.
3. **2017**: React introduced major upgrades like **React Fiber** (for better performance), error handling, and fragments (simplified code).
4. **2018**: React kept improving, adding features like Context API (for sharing data easily) and improved fragments.

**React.js** is like a tool that helps you build websites or web apps.It's a JavaScript library (a collection of pre-built functions and tools) created by Facebook to make building websites easier and faster.

### Think of React as a recipe book.

- You want to make a dish (your website), but instead of writing out every single ingredient and instruction from scratch, you use a **recipe book** (React).
- Each **recipe** in the book is like a **React component**. It tells you exactly how to make a part of the dish, like a salad or a soup.
- When you want to change the dish, like adding more salt or changing the type of bread, you don't need to rewrite the entire recipe. You just **adjust the ingredients** (components) you want to change, and the rest stays the same.

In this case, **React** is like a recipe book: it helps you organize and reuse parts of the website (like ingredients in a recipe) and allows you to quickly update just the parts you want without starting over from scratch!

## Why React Became Popular?

React solved many problems developers faced when building websites.

**Virtual DOM**:

- Before React, updating the actual webpage (DOM) was slow because the whole page had to refresh.
- React uses a **Virtual DOM**, which is like a lightweight copy of the real DOM. It figures out what's changed and updates only those parts, making the app much faster.

**Components**:

- React breaks the UI into **small, reusable pieces** called components (like a button, header, or menu).
- You can reuse these components in different parts of the app, which saves time and keeps code clean and organized.

**Declarative Code**:

- In React, you just tell it **what you want the UI to look like**, and React handles how to make it happen.
- For example, you write "Show this list of items," and React will handle adding, removing, or updating items as needed.

**Easy to Learn and Use**:

- React lets you write UI with a mix of HTML and JavaScript (called **JSX**), which feels natural for web developers. This makes it easier to build and understand.

**Strong Community and Ecosystem**:

- React is supported by Facebook and has a huge developer community. It also has many libraries and tools (like Redux, React Router) that make development easier.

## In simple way using real-life analogy

Imagine you run a **pizza shop**. Customers can order pizzas, and you have a big menu board showing all the orders.

Now, if one customer changes their pizza toppings, you wouldn't throw away the entire menu and rewrite it—you'd just update that one order.

Before React, websites worked like throwing away and rewriting the entire menu board every time something changed. This was slow and inefficient.

React became popular because:

1. **It Updates What's Needed**: React updates only the specific order (like changing just one pizza topping on the menu board).
2. **Reusable Recipes**: You can make pizzas using reusable recipes (like React components). If you already know how to make a "Pepperoni Pizza," you can use the same recipe anywhere without starting from scratch.
3. **Fast and Easy**: Customers (users) see their updated orders instantly without waiting.

**In Real Life**: React is like a super-smart system for your pizza shop—it updates only what's necessary, reuses recipes, and makes your job faster and more efficient. That's why developers love React for building websites!

## What are Hooks ?

In very simple laymen terms,Hooks are **special functions** in React that let you add extra abilities to your components.
**React Rule**: Every Hook's name must start with **use**. This helps React know you're working with a Hook.

Imagine you have a simple component (like a block of code) that can only display things. Hooks give it "superpowers," like:

1. **Remembering things** (like how many times you clicked a button).
2. **Doing something extra** (like loading data when the page opens).

## Why did React introduce Hooks?

Before Hooks, React had two types of components:

1. **Functional components**: Simple and easy to write but couldn't do much (like a plain notebook).
2. **Class components**: More powerful but harder to write and manage (like a complicated machine with lots of buttons).

As React became more popular, developers wanted to use **functional components** (because they were simpler), but they also needed the **advanced features** that only class components had (like remembering things or reacting to changes).

**Hooks** were introduced to solve this problem. They allowed **functional components** to have the same **advanced abilities** as class components but in a much simpler and cleaner way!

## Real-life analogy

Think of building a car:

- **Class components** are like having a **full-featured car**. It has **everything**: a powerful engine, advanced features like air conditioning, automatic windows, and more. But it's **harder to maintain** and **more complicated** to use.
- **Functional components** are like having a **simple car**. It gets you from point A to point B, and it's easy to manage, but it doesn't have all the fancy features like the advanced car.

Now, imagine you want to **add those extra features** (like a better engine or automatic windows) to your simple car, but you don't want the car to be too complicated.

**Hooks** are like the **upgrade kits** you can easily add to your simple car to give it the **advanced features** without making it as complex as the full-featured car!

## How Do Hooks Make React Unique?

Hooks make React unique by allowing developers to write **simpler** and **more powerful** components. Before hooks, React had to rely on **class components** for features like **state management** and **side effects**. With hooks, you can add those same powerful features to **functional components**—the simpler and cleaner way to write code in React.

### 1. Simpler Code

- Before hooks, **class components** had to be used to manage state or perform side effects (like fetching data). Class components were often **long** and **complicated**.
- With hooks, **functional components** can do all of that in **much simpler code**. You don't need to worry about writing classes, constructors, or managing a lot of boilerplate code.

## 2. Reuse Code Easily

- With hooks, you can **reuse the same logic** in different places. For example, if you need to fetch data in multiple parts of your app, you can write a custom hook and **use it wherever needed**.

## 3. Faster and More Efficient

Hooks like **useMemo** or **useCallback** help make React apps faster by **only updating the parts** of the app that really need it. This means your app can work more **efficiently** without wasting time reloading unnecessary parts.

## Types Of Hooks

### 1. Built-in Hooks

These are the hooks that come **with React**. You don't need to install anything extra to use them. They are built to solve common problems like managing state, side effects, or accessing the DOM.

**Real-life analogy**: Think of built-in hooks like tools in a toolbox that come with the toolset (React) itself. You just grab them and use them without needing anything else.

### 2. Third-party Hooks

These hooks are created by the **React community** or other developers to make common tasks easier. You install a package to use these hooks.

**Real-life analogy**: Imagine third-party hooks as getting extra tools from a hardware store that help with specific tasks, like building a fence. These tools weren't in the toolbox, but they make your job easier.

### 3. Custom Hooks

Custom hooks are **functions you create** to reuse logic across multiple components. These hooks are built using **React's built-in hooks** and can be shared with other components.

Let's go to more deep dive into **Built-in Hooks**

## useState Hook

**useState** is a special function in React that lets you create and manage changing data (called state) in functional components. It helps your component keep track of things like user input or a button click and update the screen automatically when the data changes. Each time you use useState, you create one piece of state and a way to update it. It was added in React 16.8 to let functional components handle state, which only class components could do before.

## Real-life analogy

The **useState** hook in React is like a **magic box** where you can store **something** and **keep track of it.** That "something" could be a number, text, a list, or anything you want. It's especially useful when you want your webpage to change and update automatically when this "something" changes.

## Why did React introduce useState hook?

The **useState** hook is needed in React because it allows **functional components** to handle **state** (data that changes over time) and automatically update the user interface when that state changes.

## 1. To Add State to Functional Components

- Before React 16.8, only class components could have state, which made functional components limited.
- **useState** allows functional components to track and manage changing data, making them more powerful and flexible.

## 2. To Handle Dynamic Data

- Apps often deal with data that changes, like:
  - Button clicks.
  - User input in forms.
  - Real-time updates like counters, toggles, or API responses.

- **useState** lets you store this data and ensures the UI automatically updates whenever the data changes.

### 3. To Make Components Interactive

- Without state, a React component is static—it doesn't change once it renders.
- With **useState**, components can react to user actions. e.g.show/hide element

### How do we use useState hooks?

Before using the useState hook we need to import it from React.

JavaScript
```
import {useState} from "react";
```

We can initialize different types of value such as object, array etc., to the state variable using the useState hook or it can also be initialized with null value also.

JavaScript
```
// state variable is initialized with null value

const [state, setState] = useState();

// State variable is initialized with a string

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

// state variable is initialized with a number

const [age, setAge] = useState(20);

// state variable, an array

const [names, setNames] = useState([]);

// state variable, an object

const [details, setDetails] = useState({name : Chandra , age:
21})
```

## Updating state variable :

```javascript
// Here the state variable i.e., age is initialized with a number
const [age, setAge] = useState(18) ;

// When the button is clicked, it increments the age by 1 year
<button onClick={ () => setAge(age + 1) }>
    add one year
</button>
```

## Lets understand a real-life scenario based question

### Question: Real-Time Like and Comment System

In apps like Facebook or Instagram, when users interact with a post (like it or leave a comment), the **number of likes** or **comments** is updated dynamically, and the UI reflects the changes in real-time.

### Challenges:

1. **Track Likes**: You need to keep track of how many likes a post has and whether the current user has liked it.
2. **Handle Comments**: You need to manage a list of comments and dynamically add new ones as the user submits them.
3. **Dynamic UI**: The UI must instantly update to reflect the new number of likes and display the latest comments without refreshing the page.

**Solution** GitHub Link

Code Explanation: GitHub_Link

Deployment Link Link

## useReducer Hook

### Why did React introduce the useReducer hook?

**useReducer** hook in react is used for state management. This hook is implemented in large projects where there is a need for managing multiple states.

**B**ut if **state management** will be done **useState** hook, Then why did React introduce **useReducer** Hook?

### Challenges React face with useState hook

#### 1. Too Many Separate States

- If your app needs to handle a lot of small things (like name, email, password), you need to create a separate state for each one.
- This makes your code **long and messy**. Imagine having to track 10 different things—your code would feel like juggling too many balls!

**Example:**

```JavaScript
const [name, setName] = useState("");

const [email, setEmail] = useState("");

const [password, setPassword] = useState("");
```

It's hard to keep track of all these states as the app grows. **complete**

#### 2. Scattered Logic

- Every time you want to change the state, you need to write separate update logic all over your code.
- If you have a lot of actions (like "add item," "remove item," "clear all"), the logic becomes **messy and hard to follow**.

**Example**

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

const increment = () => setCount(count + 1);

const decrement = () => setCount(count - 1);

const reset = () => setCount(0);
```

Now imagine 10 different actions—your component will be full of separate functions for each one!

To handle these problems React introduced **useReducer** to handle state updates in a more **organized and predictable** way.

## What is useReducer hook?

**useReducer** is a **React hook** that helps you manage more complex state logic. It's like a tool designed for situations where **useState** isn't enough because the state or the way it updates is too complicated.

**Syntax:**

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

1. **useReducer**:
   - This is the React hook you call to use the reducer pattern.
2. **Parameters**:
   - **reducer**: A function that defines how the state should change based on an action.
   - **initialState**: The starting value of your state.
3. **Return Value**:
   - **state**: The current value of the state.
   - **dispatch**: A function you call to trigger an action and update the state.

## What does useReducer solve?

### Keeps State Updates Organized

- Instead of writing scattered update logic (like setState) everywhere, **useReducer** puts all the logic in **one central place** called the **reducer function**.
- This makes it **easier to manage and understand**, especially when the app gets bigger.

1. Imagine a shopping list:
    - With **useState**, you'd need separate functions for adding, removing, and clearing items, all over the place.
    - With **useReducer**, you have one central list of rules (the reducer) to handle everything.

### Avoids Repeated Code

- With **useState**, you might repeat a lot of code for similar actions (e.g., increment, decrement, reset).
- **useReducer** removes this duplication by letting you define all the **"what-to-do" rules** in one place.

**Example:**

```javascript
function reducer(state, action) {
  switch (action.type) {
    case "increment": return { count: state.count + 1 };
    case "decrement": return { count: state.count - 1 };
    default: return state;
  }
}
```

Let’s understand a real-life Problem

## Form Validation with Reducer

Create a multi-field form (e.g., name, email, and password) using useReducer where:

- Each input field updates the state via a reducer.
- Validate the fields on form submission (e.g., email must include "@" and password must be 8+ characters long).
- Show an error message if validation fails.

**Challenge:** Write the reducer to handle both field updates and validation logic.

Solution: GitHub_Link

Code Explanation GitHub_Link

Deployment Link Link

## useEffect Hook

### Why did React introduce useEffect hooks?

React introduced the **useEffect** hook to solve problems and make life easier for developers when managing **side effects** in functional components.

challenges before **useEffect hook**

### 1. Class Components Were Complicated

Before React introduced hooks (like useEffect), you could only handle side effects using **class components**. In class components, you had to use special lifecycle methods like:

- **componentDidMount** (when the component first shows up)
- **componentDidUpdate** (when the component updates)
- **componentWillUnmount** (when the component is removed)

Here the Challenges are:

### 1. Code was harder to read and maintain:

- You had to split the same logic across different lifecycle methods.
- It was easy to make mistakes if you forgot to handle something in the right lifecycle method.

### 2. Duplicate code:

- If you wanted to do the same thing in componentDidMount and componentDidUpdate, you'd end up writing the same logic twice.

### 2. No lifecycle methods in functional components:

- If you wanted to use side effects, you **couldn't use functional components**. You were forced to use class components, even if you didn't need them.

To solve these problems React introduced **useEffect Hook.**

1. **Works on functional components.**
2. **Combine lifecycle methods into one place.**
3. **Easier cleanup.**

## What is useEffect hook:

The useEffect hook in React is like a helper that lets your component **do something extra** after it renders. You can think of it as a tool that handles things like:

1. **Fetching data** from a server (e.g., grabbing user info or a list of items).
2. **Updating the webpage** (e.g., changing the page title).
3. **Cleaning up** when the component is no longer needed (e.g., stopping a timer).

Let understand it's more simple way relating to real-life analogy

👉 **"Do something extra after showing the page!"**

Imagine you're making tea:

1. First, you **boil water** (React shows the page – this is "rendering").
2. Then, you **add tea leaves** after the water is boiled (this is what **useEffect** does – the extra work after the page is ready).

## What are the Features of useEffect hooks?

### 1. Runs After the Page Renders

When the page (or component) shows up on the screen, **useEffect** can automatically run some code **after the rendering is done**.
This is helpful for doing things like fetching data, updating the title, or starting a timer.

```javascript
useEffect(() => {
  console.log("Hello");
});
```

## 2. Runs When Something Changes

You can tell **useEffect** to run only when certain things (called **dependencies**) change.
This helps React know when to do the "extra work."

```javascript
useEffect(() => {
  console.log("Count changed!");
}, [count]); // Runs only when `count` changes
```

## 3. Runs Only Once (Optional)

If you want **useEffect** to run **just one time** (e.g., when the page loads), you can give it an **empty array** ([]).
This is great for tasks like fetching data or setting something up when the page starts.

```javascript
useEffect(() => {
  console.log("run once");
}, []); // Runs only once
```

## 4. Cleans Up After Itself

If you start something (like a timer or an event listener), **useEffect** can clean it up when the component is removed or updated.
It does this using a **cleanup function** inside the return statement.

```javascript
useEffect(() => {
  const timer = setInterval(() => {
    console.log("Running a timer...");
  }, 1000);

  // Cleanup when the component is removed
  return () => {
    clearInterval(timer);
    console.log("Timer stopped!");
  };
}, []);
```

## Lets understand a real-life scenario based question

## Like Button Update with Real-Time Notifications

**Scenario:** In an Instagram-like app, when a user likes a post, you need to:

- Update the like count in real-time.
- Display a notification to the user that their like has been successfully recorded.
- Clean up the notification once the user views it.

### Challenges:

- Uses **useEffect** to listen for the like button press and update the like count.
- Show a notification when the like is successfully recorded.
- Dismiss the notification after 3 seconds.

**Solution** GitHub Link

**Code Explanation** GitHub_Link

**Deployment Link** Link

## useMemo Hook

### Why did React introduce the useMemo hook?

Before **useMemo** was introduced, React had no built-in way to **remember** or **skip** certain calculations during re-renders, especially when those calculations were expensive or slow.

### Challenges faced:

**Unnecessary re-calculations:** Every time a component re-rendered (for example, when state or props changed), React would re-run the entire function, including expensive calculations like sorting a large list or making complex math calculations. Even if the data didn't change, React still re-ran those calculations, which slowed down the app.

**Performance issues:** As the app grew more complex and the components became more dynamic, React was re-running expensive code repeatedly, causing **performance bottlenecks**. This could make apps feel slow and unresponsive.

So to solve this problem, React introduced **useMemo** to **optimize performance**. With **useMemo**, React can **remember** the result of an expensive operation and only re-run it when the relevant data changes. This prevents unnecessary calculations and speeds up the app.

### What is useMemo hook?

**useMemo** is a React Hook that allows you to **memoize** (i.e., store) the result of an expensive function call and only recompute it when specific dependencies change. It helps optimize performance by preventing unnecessary recalculations on every render, which can be particularly helpful for computationally expensive operations or large data sets.

In a very simple way, imagine you're doing a task, like adding up numbers. Normally, you'd add them every single time you need the result. But what if the numbers don't change? You don't need to keep adding them over and over. Instead, you can just **remember the result** and use it again.

In React, useMemo helps you **remember** the result of a slow task (like a calculation or sorting) and **reuse it** without doing it again, unless something important changes.

So, **useMemo** saves you from doing the same slow work repeatedly, making your app faster. It remembers the result and only updates it if needed.

## What are the advantages of useMemo hook?

### 1. Makes Your App Faster

- **useMemo** helps your app run faster by **remembering** the result of slow tasks (like sorting or calculating) so that it doesn't redo the same work over and over again. This way, your app doesn't waste time on tasks that haven't changed.

### 2. Avoids Unnecessary Updates

- If something in your app doesn't change (like a number or list), **useMemo** makes sure React **doesn't re-calculate** or re-render parts of your app. This means React only works on things that **really need to change**.

### 3. Keeps Expensive Calculations in Check

- If you have slow or heavy tasks (like sorting a large list), **useMemo** ensures React **only runs those tasks when needed**. It saves time and makes the app faster by reusing results when nothing changes.

### 4. Smoothens User Experience

- By making calculations faster and avoiding unnecessary work, **useMemo** helps your app feel more **responsive**. Your app won't freeze or slow down when updating the screen, leading to a better experience for the user.

### 5. Helps with Big Data

- When your app handles large amounts of data (like big lists or objects), **useMemo** helps by **saving the results** of any slow calculations. This means React won't keep recalculating the same things over and over.

### 6. Saves Memory in Some Cases

- If a calculation uses a lot of memory, useMemo can help by **storing** the result and reusing it, rather than repeating the whole process every time.

## When to Use useMemo?

### 1. Expensive Computation

If your component is doing something that takes time, like working with large amounts of data (e.g., filtering, sorting, or calculations), **useMemo** can help. It remembers the result so it doesn't have to redo the work every time the component updates.

### 2. Reusing Derived Data

When your component derives some data from props or state and recalculating this derived data is expensive, you can use **useMemo** to avoid unnecessary recalculations.

- **Example**: Filtering a list of products based on a search query or transforming large amounts of data.

```javascript
const filteredProducts = useMemo(() => {
  return products.filter(product =>
product.name.includes(searchQuery));
}, [products, searchQuery]);
```

### 3. Preventing Unnecessary Re-renders of Child Components

If you pass a calculated value (like a filtered list or sorted data) as a prop to a child component, React might re-render the child

unnecessarily. Use **useMemo** to ensure that the value only changes when necessary.

**Syntax:**

```javascript
const memoizedValue = useMemo(() => {
  // Expensive operation or calculation
  return result;
}, [dependencies]);
```

- **memoizedValue**: The value that will be memoized and reused in future renders.
- **useMemo**: The hook used to memoize the value.
- **() => { ... }**: A function that performs an expensive operation or calculation.
- **[dependencies]**: An array of values (dependencies) that, when changed, will trigger a re-calculation of the memoized value.

**Lets understand a real-life scenario based question how useMemo work**

**Fitness App Progress Tracking**

Scenario:
You're creating a fitness app where users can track their daily workout progress (e.g., total calories burned, hours worked out). Aggregating and displaying trends over time is computationally expensive.

Task:

- Use **useMemo** to calculate aggregated stats and trends.
- Ensure recalculations only happen when workout data changes.

Solution GitHub Link

Code Explanation GitHub Link

Deployment Link Link

## useCallback Hook

### Why did React introduce useCallback hook?

React introduced the **useCallback** hook to address specific challenges related to **performance optimization** and **unnecessary re-rendering** in React applications.

### Challenges that developers face

Before **useCallback**, developers faced certain issues while working with **functions inside functional components**, especially when passing functions as props to child components.

**1. Unnecessary Function Recreation**

In React, every time a functional component re-renders, all the functions defined inside that component are **recreated**. This includes functions that are passed as props to child components.

When functions are recreated on every render, even if their logic hasn't changed, they trigger re-renders in child components unnecessarily.

### Key Reason: React Checks Using ===

React uses a simple "is it the same?" test (called **shallow equality**) to check if props have changed. It looks at memory references:

- If the function **reference** is different, React thinks it's a brand-new function, even if the function's logic hasn't changed.
- This triggers the child component to re-render.

Let's understand through **some real-life analogy**

Every day, the teacher **writes a brand-new homework assignment** (function) for the students, even if the task is exactly the same as yesterday.

1. The teacher writes the homework again, even though it hasn't changed.
2. The students get the new homework every day and think, "Oh, this is a new task!" and start doing it from scratch.

3. Even if the task was exactly the same as the day before, the students waste time thinking it's different.

Even though the task (function) didn't change, it **looks different** every day to the students. This makes them **do extra work** unnecessarily.

### With useCallback:

Now, imagine the teacher has a **pre-written homework assignment** (function) that stays the same unless the task actually changes.

1. The teacher **gives the same homework** every day.
2. The students see it's the **same** homework they got yesterday, so they don't waste time thinking it's new.

### What is useCallback hook?

The **useCallback** hook is a tool in React that helps you **avoid recreating the same function** every time your component re-renders.

### Advantages of useCallback hook

1. **Avoids Unnecessary Re-renders:**
   - React re-renders components when their props change. If you pass a new function (even if it's the same logic), React thinks it's different and may re-render child components unnecessarily.
   - **useCallback prevents this** by "remembering" the function, so it stays the same unless something changes.

2. **Improves Performance:**
   - Every time React creates a new function, it has to do extra work. By using **useCallback**, you can avoid that **extra work**, which can improve your app's speed—especially if there are many components or expensive operations happening.

**Syntax:**

```javascript
import { useCallback } from 'react';


const YourComponent = () => {
  const memoizedFunction = useCallback(() => {
    // Your function logic here
  }, [dependencies]);


  return (
    // Your JSX code here
  );
};
```

**useCallback** takes two arguments:

1. **The function**: This is the function you want to remember and avoid recreating on each re-render.
2. **The dependency array**: This is a list of values (like state or props) that React will check. If any of these values change, it will create a new version of the function. If they don't change, it will use the **same function**.

Let's understand through a real-life example

**Stopwatch with Start, Stop, and Reset**

**Task**:

- Create a stopwatch with the following features:
  - A button to **start** the timer.
  - A button to **stop** the timer.
  - A button to **reset** the timer.

- Use **useCallback** to memoize the start, stop, and reset functions to avoid recreating them unnecessarily.

**Challenge**:

- Keep the timer running efficiently without triggering unnecessary re-renders of the buttons or display.

Solution:

GitHub_Link

Code Explanation:

GitHub_Link

Deployment Link

Link

## useRef Hook

### Why was useRef introduced?

React introduced **useRef** to solve **two problems**:

1. **Accessing a DOM element (like a button or input box) easily.**
   Before, it was hard to directly control elements, like focusing an input box or scrolling a div, in functional components.
2. **Storing data without causing re-renders.**
   If you used useState to save data, React would re-render the component every time the data changed. Sometimes you just want to save something without re-rendering.

### Challenges react faced before using useRef hook

- **Messy DOM access**
  React used to rely on older tools like **React.createRef**, which only worked in class components and were harder to use.
- **Too many re-renders**
  Saving data in **useState** would unnecessarily re-render the component even if you didn't want it to.

### What is the useRef hook?

**useRef** is a **React hook** that lets you create a **reference object**. This reference object:

1. Has a property called **current**.
2. You can use **current** to **store a value or reference a DOM element**.
3. The value inside **current** is **mutable** (you can change it).

### Key Features

- **Doesn't trigger re-renders**: If you change the value inside **current**, React doesn't re-render the component.
- **Value persists across renders**: The value inside **current** is not reset when the component re-renders.

**L**et's understand it very simple way

**useRef** is like creating a **box** where you can store something (a value or a DOM element).

- This box is called a **ref object**.
- The ref object has a **current property**, which holds the value you want to store.
- You can **change the value inside current** or **read it whenever you need**.

The special thing is:

- The value in current doesn't disappear when the component updates.
- Changing the value in current **doesn't cause the component to re-render**.

## Why do we use useRef hook?

- **To directly access a DOM element**
  Example: If you want to focus on an input box or scroll to a part of the page, useRef helps you do it easily.
- **To store a value that doesn't need re-rendering**
  Example: If you want to track something (like a timer or a count) without updating the UI, you can store it in useRef.

**Syntax:**

```javascript
const refName = useRef(initialValue);
```

1. **refName**: The variable name you choose to store the ref object.
2. **useRef**: The hook that creates the ref.
3. **initialValue**: The starting value of the current property in the ref object (can be null, a number, a string, etc.).

Let's understand through a real-life example

## Focus Management in a Form

**Task**:

- Create a form with multiple input fields (e.g., Name, Email, Password).
- Use **useRef** to focus on the next input field when the user presses "Enter" in the current field.

**Challenge**:

- Implement smooth navigation between inputs without re-rendering the entire form.

Solution:

GitHub_Link

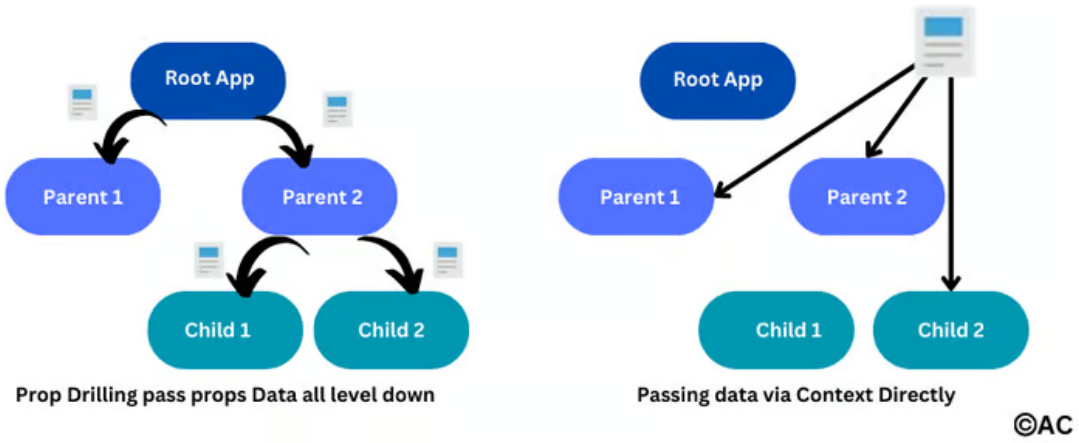
Code Explanation:

GitHub_Link

Deployment Link

Link

## useContext hook




## Why did React introduce useContext hook?

React introduced **useContext** to solve the problem of **"prop drilling"** and to make sharing data between components much easier.

## What is prop drilling?

Before **useContext**, if you wanted to share data with a deeply nested component, you had to pass it through every component in between, even if those middle components didn't need the data.

**Example:**

Let's say you have a **Parent** component, and inside it, there's a **Child**, and inside that, there's a **Grandchild**. If the **Grandchild** needs some data from the **Parent**, the **Child** also has to pass that data, even if it doesn't use it. This is called **prop drilling**, and it makes the code messy and hard to maintain.

## Challenges React faced before useContext

- **Messy code**: Passing props through many levels of components made the code harder to read and understand.
- **Hard to maintain**: If you needed to change something, you had to update every component in the chain.

- **Unnecessary work**: Components in the middle (like the **Child**) had to handle data they didn't even need, just to pass it down.

**To** handle such type of error React introduced **Context api**

React introduced **useContext** so that any component can directly access shared data without passing it through every level. This makes the code cleaner and easier to manage.

Let's understand it more simple way

Before **useContext**, it was like giving a letter to one person (Parent), asking them to pass it to another (Child), who then passes it to the final person (Grandchild). This was tiring and unnecessary.

With **useContext**, it's like putting the letter on a shared table (context), and whoever needs it (like the Grandchild) can pick it up directly. No need to bother the Parent or Child anymore!

In short: **React introduced useContext to avoid passing data through every component (prop drilling) and make sharing data simple and direct.**

## What is Context api?

The **Context API** in React is a tool that lets you share data between components **without passing props manually** through every level of the component tree.

In simple terms,The **Context API** is like a **common storage box** where you put data, and any component in your app can take the data directly from this box instead of passing it around through props.

## Why Use the Context API?

**Avoid Prop Drilling**: You no longer need to pass props manually down every level of the component tree.

**Global State Management**: Context allows you to manage and share state across your application easily, without relying on external libraries like Redux.

**Improved Code Organization**: The Context API keeps your code cleaner by eliminating unnecessary prop-passing, especially in larger apps.

## Syntax:

### 1. Create a Context

First, you create a context using **React.createContext()**.

```javascript
import React, { createContext } from "react";


const MyContext = createContext(); // This creates the context
```

### 2. Provide the Context

Wrap the components that need access to the context with the **Provider**. The **Provider** shares the data with its children.

```javascript
<MyContext.Provider value={sharedData}>
  <YourComponent />
</MyContext.Provider>
```

- **value** is the data you want to share (like a string, object, or anything).

### 3. Consume the Context

Use the **useContext** hook to access the context data in any component.

```javascript
import React, { useContext } from "react";


const contextValue = useContext(MyContext); // Access the
shared data
```

**L**et's understand through a real-life question

## User Authentication

**Question:**
Build a React app that displays different content for logged-in and logged-out users using the useContext hook.

**Hint:**

- Create an **AuthContext** with the user's authentication status (isLoggedIn).
- Provide a function to log in and log out.
- Use **useContext** in different components to show either "Login" or "Welcome, User!"

**Solution**

GitHub_Link

Code Explanation:

GitHub_Link

Deployment_Link
Link

## useSelector Hook

The useSelector hook was introduced by React-redux library.

Before going to useSelector hook directly, let's understand some basic knowledge of **why did Redux-toolkit introduce it?**

## What is redux?

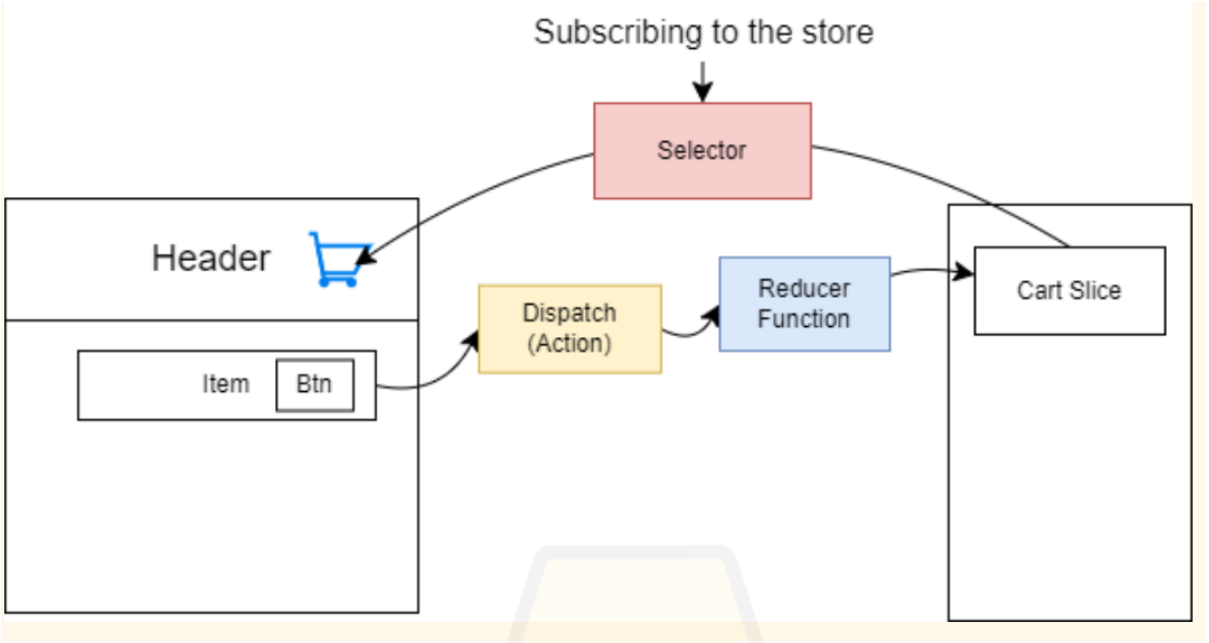
**Redux** is a **state management library** for JavaScript applications, often used with frameworks like **React**. It helps to connect **React components** to a **centralized store** (where the app's data is stored) and makes sure the data is consistent and easy to manage.

**Why Redux was introduced (short and simple):**

1. **Props Drilling is Messy**
   - Passing data through many components (e.g., App → Dashboard → Profile → Avatar) is annoying and hard to manage.
   - **Redux solution**: Directly gives data to the components that need it.
2. **Context API is Limited for Large Apps**
   - **Performance issues**: Context can cause many unnecessary updates, slowing down big apps.
   - **Hard to debug**: In big apps, finding the root cause of a problem is tricky with Context.
   - **Redux advantage**:
     - Prevents unnecessary updates.
     - Includes powerful debugging tools (Redux DevTools) to track every change easily.

In short, **Redux keeps things organized and efficient**, especially for large apps.

How does Redux work?




## What is the useSelector hook?

The **useSelector** hook in React is like a **shortcut to grab data** from a store (usually a Redux store). Imagine you have a big bag of data (Redux store) that everyone in your app can access. Instead of digging into the bag manually every time, **useSelector** lets you pick exactly what you need from that bag.

In simple terms, imagine your app has a **big fridge** (Redux store) that stores all the ingredients (data) your app needs.
Instead of taking out the whole fridge, **useSelector** lets you open the fridge and take only the specific ingredient you need (like milk or eggs).

## Why do react use useSelector hook?

React needs the **useSelector** hook because it helps React **know what data to use** from a central storage place (Redux store). Without it, React wouldn't know how to get specific data that your app needs to show.

**Syntax:**

```javascript
const selectedData = useSelector((state) => state.someData);
```

### Explanation of the Syntax:

1. **useSelector**: The hook provided by React-Redux to access data from the Redux store.
2. **Callback Function (state) => state.someData**:
   - The state parameter represents the entire Redux store.
   - You return the specific part of the store you want, like state.someData.
3. **selectedData**:
   - This is the variable where the selected data is stored.

Let's understand real-life example

### Getting a List of Products

The Redux store contains a products array. Use the **useSelector** hook to display the names of all products.

Store structure:

```javascript
{
  products: [
    { id: 1, name: "Laptop", price: 800 },
    { id: 2, name: "Phone", price: 500 },
    { id: 3, name: "Tablet", price: 300 }
  ]
}
```

**Solution:**

GitHub_Link

Code Explanation
GitHub_Link
Deployment Link
Link

## useDispatch Hook

The useDispatch hook was introduced by react-redux library.
In React with Redux, the **useDispatch** hook is used to **send actions** to the Redux store. An action is just an object that describes **what you want to do**, like updating some data in the store.

## Why do we need useDispatch?

The Redux store doesn't know what changes you want to make until you send it an **action**. useDispatch gives you the **function** that lets you send these actions to the store.

### Syntax

1. Import useDispatch from react-redux

```javascript
import { useDispatch } from 'react-redux';
```

2. Get the **dispatch** function:

```javascript
const dispatch = useDispatch();
```

- Creates the **dispatch** function for sending actions.

3. Use **dispatch** to send an action

```javascript
dispatch({ type: 'ACTION_TYPE', payload: { key: 'value' } });
```

Sends an action object to the Redux store:

- **type**: Describes what to do.
- **payload**: Optional data for the action.

Let’s understand a real-life example

You are building a **grocery shopping list app** using React and Redux. The app allows users to add items to the list.

Write a React component where:

1. Users can type the name of a grocery item in an input field.
2. When they click the "Add Item" button, the item is added to the list using the **useDispatch** hook.

**Solution:**

GitHub_Link

Code Explanation

GitHub_Link

Deployment Link

Link

## useStore hook(Zustand)

### What is Zustand?

**Zustand** is a tool that helps you manage data (called "state") in your React apps in a simple and flexible way.

Imagine you have a box where you keep your app's important information (like numbers, text, or objects). Zustand lets you easily create this box (called a **store**), put the data in it, and change or read the data when needed.

### Why does react use zustand?

React developers use **Zustand** because it offers a **simple, flexible, and lightweight** way to manage state in their applications. It allows developers to handle state more easily without the complexity that comes with other state management libraries like **Redux**.

**1. Simplicity**

- Zustand is incredibly simple to set up and use. You don't need to write complex boilerplate code like in Redux.
- You can directly create a store, define state and actions in a single place, and access them using hooks.

**2. No Context or Provider Needed**

- Zustand does not require wrapping your app in a **Provider** component, as it uses React hooks directly. You can use the store anywhere in your app without needing additional setup.

**3. Lightweight & Fast**

- Zustand is designed to be **lightweight**, making it ideal for small to medium-sized applications. It doesn't introduce any unnecessary overhead and is **very fast**.

**4. Good for Small and Medium Apps**

- Zustand shines in apps where you don't need the full power of Redux but still need reliable state management. It's a perfect

choice for small to medium React apps where simplicity and speed matter more than structure.

Then **why do developers use Redux-toolkit?**

- Developers often choose **Redux Toolkit** because it provides a **structured, predictable, and scalable solution** that works well in large applications with complex state management needs, asynchronous operations, and enterprise-level requirements.

## What is useStore hook?

Imagine you have a **box** (a store) where you keep your important stuff (like numbers, text, or any data). The **`useStore`** hook is like a **key** that lets you open that box to see what's inside or put new things in it.

### How does it work?

- You **create a store** (a box) to hold some data.
- You **use the `useStore` hook** to open the box and either **look at the data** or **change it**.

Let's understand it through simple problem

### Simple Form Input

- **Problem**: You have a form with a text input field, and you want to store the input value globally, so other parts of the app can access it.
- **hints**: Use **useStore** to store the input value and update it when the user types.

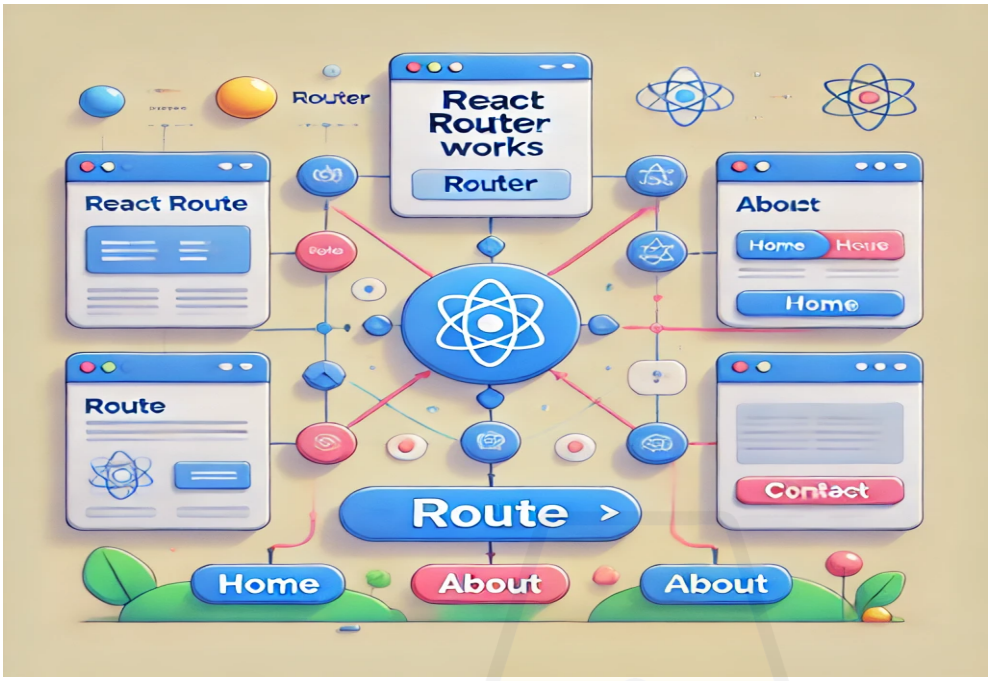
Solution:

GitHub_Link

Code Explanation GitHub_Link

Deployment Link Link

## React-router Dom




**React Router is a library** for handling routing and navigation in React JS Applications.

### Why did React need React-routing?

React needs routing to handle **navigation** between different parts (or pages) of your website in a smooth way.

**B**efore React introduced routing and other features,

### There are some Challenges react face:

**1. Slow Page Reloads:**

Every time you clicked on a link, the entire page would **reload**. It made the website feel slow and **annoying** because you had to wait for everything to load again.

**Problem**: Imagine you're watching a video and every time you click on something, the video stops and the whole page reloads.

**2.** **Losing Data**

Without routing, switching between pages could cause **data loss**. For example, if you had a form filled out, it would reset every time you clicked a new page.

**Problem**: You filled out your details in a form, but when you clicked "next," everything got wiped out.

**3.** **Difficult to Reuse Code:**

Before routing, React didn't have a good way to handle different parts (or pages) of the app. You had to write extra code to make everything work smoothly, and it was hard to **reuse** parts of the app.

**Problem**: Every time you wanted a new page, you had to rewrite parts of your app instead of just reusing them.

To overcome these problems React needs React-router-dom library

1. **No More Page Reloads**: React Router lets you change pages smoothly without reloading the whole website, making the app feel faster.
2. **Reuses Code**: You can create a **single header or footer** and use it on all pages, instead of writing the same code over and over.
3. **Manages Navigation Easily**: It provides an easy way to set up and manage different pages in your app without complicated coding.

In short, React Router DOM makes navigation **easy, fast, and organized** by handling page changes without reloads, updating URLs, and reusing components.

## useNavigate Hook

**useNavigate** hook is part of the **React Router library**.it gives you a function that can programmatically change the current location in your app (i.e., navigate to a different URL). It provides a way to navigate without needing to use traditional **<Link>** elements or **a** tags.

In simple terms,Imagine you're in a big house (your app) with different rooms (pages). To move between rooms, you usually open the doors (links) by clicking on them.

But sometimes, you need to move to a new room (page) without clicking on a door. Instead, you want to just tell the app, "Go to this room" at the right moment (like after clicking a button). That's where **useNavigate** comes in!

## How does the useNavigate hook work?

**1.** **you first need to install React Router**

```
JavaScript
npm install react-router-dom
```

**2.** Import **useNavigate**

```
JavaScript
import { useNavigate } from 'react-router-dom';
```

**3.** Get the **navigate** function

```
JavaScript
const navigate = useNavigate();
```

**4.** Use the **navigate** function

Use **navigate()** to navigate to a different route. You can pass it a string representing the path, or you can configure other options.

```javascript
navigate('/path');  // Navigate to a specific path
```

Let’s understand through simple real-life Problem

## Redirect to the homepage after login

- **Problem:** Create a simple login page where, after entering the correct username and password, the user is redirected to the homepage.
- **Hint:** Use **useNavigate** to navigate to the homepage once the login is successful.

Solution:

GitHub_Link

Code Explanation

GitHub_Link

Deployment Link

Link

## useParams Hook

**useParams** hook in React router library is like a helper that lets you get the **dynamic parts** of the URL in your component.

For example, if your URL is something like **/user/123**, the number **123** is a **dynamic part** that can change. Using useParams, you can easily get that part of the URL (in this case, 123) inside your component and use it.

### How does the useParams hook work?

**1.** **you first need to install React Router**

```JavaScript
npm install react-router-dom
```

**2.** Import **useParams**

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

**3.** **Set up a route with a parameter:** You define a route with something like **/user/:id**, where **:id** is a placeholder for the dynamic part.

```JavaScript
<Route path="/user/:id" component={UserPage} />
```

**4.** **Use useParams to get the value of id:** In the UserPage component, you use useParams to access that dynamic part (id).

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


function UserPage() {
```

```
  const { id } = useParams();  // 'id' is the dynamic part of
the URL
  return <h1>User ID: {id}</h1>;
}
```

if the URL is **/user/123**, **useParams** will give you **id = 123.**

## Let's understand it through a real-life simple problem

Create a React component for a cooking website where users can view a recipe based on its unique ID from the URL (e.g., /recipe/:recipeId). Use the **useParams** hook to extract the recipe ID from the URL and fetch the recipe details from a mock API. Display the recipe name, ingredients, and instructions on the page.

Solution:

GitHub_Link

Code Explanation

GitHub_Link

Deployment Link

Link

## useLocation Hook

The **useLocation** hook in React Router is used to return the current location of a React component.

Imagine you are on a website, and the URL is something like **"www.xyz.com/about".** The **useLocation** hook helps you know that you're on the "about" page, by giving you the current location (the full URL or just the path) you're at in the app.

It returns an object with information like:

- **pathname**: The path part of the URL (like **/about**).
- **search**: Any query parameters (like **?search=react**).
- **hash**: Any part of the URL that comes after a # (like **#section1**).

## How does the useLocation hook work?

1. **you first need to install React Router**

```
JavaScript
npm install react-router-dom
```

2. Import **useLocation**

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

3. Use the **useLocation** hook inside a component

```
JavaScript
const location = useLocation();
```

4. Access location properties

```
JavaScript
import React from 'react';

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


function MyComponent() {
```

```
  const location = useLocation();

  return (
    <div>
      <h1>Current Path: {location.pathname}</h1>
      <p>Query Params: {location.search}</p>
      <p>Hash: {location.hash}</p>
    </div>
  );
}
```

Let’s understand through a simple real-life problem

## Dynamic Navigation Highlight

**Problem:**
You are building a navigation bar for a website. The active menu item should be highlighted based on the current URL.

**Task:**
Use the useLocation hook to determine the current route and add an active class to the corresponding menu item.

**Example:**

- URL: /about
- Highlight the "About" menu item.

Solution: GitHub_Link

Code Explanation GitHub_Link

Deployment Link Link

## useSearchParams Hook

### What are query parameters?

Query parameters are a way to pass data from one page to another in a URL.

**For Example**

```javascript
https://xyz.com/search?q=react&lang=en
```

In this URL, **q and lang** are **query parameters**. The value of q is react and the value of lang is en. Query parameters are often used to filter or sort data in a web application.

Now Let's understand about ,

### What is useSearchParams hook:

The **useSearchParams** hook in React is like a tool that helps you work with the "extra information" in the URL after the **?** symbol. This extra information is called **query parameters**.

Let's understand with simple analogy

You're at a pizza shop. The shopkeeper asks you for extra details about your order, like:

- What size pizza do you want? (e.g., **medium**)
- What toppings do you want? (e.g., **pepperoni**)

This information is like the **query parameters** in the URL. It tells the shop what you want.

Now, **useSearchParams** is like a notebook where:

1. You **read** what the customer wrote (e.g., "medium pizza with pepperoni").
2. You **update** it if the customer changes their mind (e.g., "change it to large pizza with mushrooms").

## How it works in a URL:

- The customer's choices are written in the URL: www.pizzashop.com?size=medium&topping=pepperoni
- With **useSearchParams**, you can:
  - **Get the details**: "size=medium" and "topping=pepperoni."
  - **Update the details**: Change it to "size=large" or "topping=mushrooms."

## Advantages

The **useSearchParams** hook in React (from the react-router-dom library) helps you work with query parameters in the URL, like **?name=chandra&age=25.**

- **Easier URL Handling**: It makes reading and updating query parameters in the URL very easy without manually writing a lot of code.
- **Real-Time Updates**: When you update the URL's query parameters, the page automatically reflects those changes without refreshing.
- **Cleaner Code**: Instead of writing a lot of code to handle query parameters, the hook gives you simple functions to get and set them quickly.

## How do we use it?

You're building a search page where users can search for products, and their search term (like "shoes") will appear in the URL. For example:

`www.mystore.com?search=shoes`

### 1. you first need to install React Router

```javascript
npm install react-router-dom
```

**2.** .Import **useSearchParams**

```javascript
import { useSearchParams } from 'react-router-dom';
```

### 3. Set up the Hook

Use the hook in your component. It gives you two things:

- searchParams → to **read** the current query parameters.
- setSearchParams → to **update** or change the query parameters.

```javascript
const [searchParams, setSearchParams] = useSearchParams();
```

### 4. .Read Query Parameters

You can use **searchParams.get()** to read the values in the URL.

```javascript
const searchQuery = searchParams.get("search"); // Get the 'search' value from the URL

console.log(searchQuery); // For example, this will log "shoes" if the URL is ?search=shoes
```

### 5. Update Query Parameters

To change or add query parameters, use **setSearchParams.**

Example: When the user types "jackets" in the search bar, you can update the URL like this:

```javascript
const handleSearch = (newSearch) => {

  setSearchParams({ search: newSearch }); // Update the URL to ?search=jackets

};
```

Let’s understand through a simple real-life problem

## Search Filter with Query Params

**Problem:**
You're building an e-commerce site and want to filter products based on categories (e.g., ?category=electronics) and sort them (e.g., ?sort=price_asc).

**Task:**
Use useSearchParams to read the category and sort query parameters and apply corresponding filters and sorting to the product list.

**Solution**

GitHub_Link

Code Explanation

GitHub_Link

Deployment_Link

Link

## useRoutes Hook

The **useRoutes** hook in React Router is like a **map or guide** for your app to decide which page (or component) to show based on the current URL. Think of it as a helper that connects your app's routes (URLs) with the components you want to show.

### How does it work?

**Define Routes:** You create a list of all your routes (URLs) and say which component should appear for each route.

**Hook into the Routes:** The **useRoutes** hook takes that list and matches it to the current URL. It then tells React which component to display.

Why use **useRoutes**?

- It makes routing **simpler and cleaner** in larger apps.
- You can manage all your routes in one place instead of writing **Route** components everywhere.

### How do we use it?

**1. you first need to install React Router**

```javascript
npm install react-router-dom
```

**2.** Import **useRoutes**

```javascript
import { useRoutes } from 'react-router-dom';
```

**3. Set Up Your Routes**

Now, go to your App.js file and define the routes you want for your app.

1. Create an array of routes with the path (URL) and the component (element) to show.

JavaScript

```
import React from "react";
import { useRoutes } from "react-router-dom";

function App() {
  // Step 3: Define your routes
  const routes = [
    { path: "/", element: <Home /> },
    { path: "/about", element: <About /> },
    { path: "/contact", element: <Contact /> },
  ];

  // Step 4: Use useRoutes to match the current URL with a
route
  const element = useRoutes(routes);

  // Step 5: Return the matched component
  return element;
}

// Dummy Components
function Home() {
  return <h1>Welcome to Home Page</h1>;
}

function About() {
  return <h1>About Us</h1>;
```

```
}


function Contact() {
  return <h1>Contact Us</h1>;
}
export default App;
```

Let’s understand through a simple real-life problem

## Dynamic Route Rendering Based on User Role

**Problem**:
You are building a multi-role web application (e.g., admin and user). Different roles should have access to different routes. Admins should have access to admin-only routes, while regular users should not. You need to render routes dynamically based on the user's role.

**Task**:
Use the **useRoutes** hook to conditionally render different sets of routes depending on the role of the user.

Solution:
GitHub_Link

Code Explanation

GitHub_Link

Deployment_Link
Link

## React Form Hooks

React developers use libraries like **React Hook Form** because, while React can handle forms manually using its core features (like state and event handlers), managing forms in more complex applications becomes tricky.

### 🚧 Challenges with Core React for Forms

1. **Handling Multiple Fields**:
   Managing state for every input manually can become messy as the number of fields grows.
2. **Complex Validations**:
   Writing custom validation logic for each field can lead to a lot of repetitive code.
3. **Form Re-Renders**:
   React re-renders the entire component every time state updates, which can slow down your app for big forms.
4. **Error Handling**:
   Displaying error messages and updating the UI dynamically (e.g., highlighting invalid fields) requires additional code.

### 🛠️ React Hook Form Solves These Challenges:

- Automatically tracks input values and updates them efficiently.
- Provides built-in validation and error handling.
- Improves performance by reducing unnecessary re-renders.

## useForm Hook

The **useForm** hook is a special tool provided by the **React Hook Form** library. It helps you manage forms in a simple and efficient way without writing too much code.

### What Does useForm Do?

It acts like a form "manager" that:

1. Keep track of your form's input values.
2. Handles form submission for you.
3. Validates the inputs (like checking if fields are empty or if an email is valid).
4. Helps you show error messages easily.

### What are the advantages of useForm hook?

The **useForm** hook is super useful because it makes managing forms in React **easier, faster, and more efficient**.

#### 1. Simplifies Form Handling

Without **useForm**, you'd have to write a lot of code to track input values, handle state changes, and manage form submissions manually. With useForm, all of this is handled for you automatically.

#### 2. Built-in Validation

You can add validation rules (like "required" or "must be an email") directly to your inputs. If the input doesn't meet the rules, **useForm** will let you know and help you show error messages.

Example:

```JavaScript
<input {...register("email", { required: "Email is required"
})} />
```

### 3. Improves Performance

**useForm** only updates the specific input fields that change, instead of re-rendering the whole form. This makes your app faster, especially if you have large forms.

### 4. Handles Form Submission Easily

**handleSubmit** takes care of what happens when you submit the form. You don't need to write a lot of logic for collecting data—it just works.

How to Use **useForm**?

### 1. you first need to install React Form

```
JavaScript
npm install react-hook-form
```

### 2. Import **useForm**

```
JavaScript
import { useForm } from 'react-hook-form';
```

### 3. Set It Up:

Use **useForm()** in your component to create a "form manager."

```
JavaScript
const { register, handleSubmit } = useForm();


const onSubmit = (data) => {
  console.log(data); // Form data will be here
};


return (
  <form onSubmit={handleSubmit(onSubmit)}>
```

```
    <input {...register("name")} placeholder="Enter your name"
/>
    <input {...register("email")} placeholder="Enter your
email" />
    <button type="submit">Submit</button>
  </form>
);
```

- **register**: This connects your form inputs to the form manager.
- **handleSubmit**: This handles what happens when the form is submitted.
- When the user types into the inputs, **useForm** automatically tracks their values.
- When you click the "Submit" button, handleSubmit takes care of submitting the form and passes the form data to onSubmit.

## Let's understand through a simple real-life problem

### User Registration Form

**Problem**:
You are building a user registration form where users can input their name, email, and password. The form should validate that all fields are filled out and that the email is in the correct format. The password should have a minimum length.

**Task**:
Use **useForm** to manage the form state and perform the necessary validation.

Solution: GitHub_Link

Code Explanation GitHub_Link

Deployment_Link Link

# React Formik for Form Handling

## Introduction:

Creating forms in React can get tricky because there are a lot of things to manage, like checking if the user entered correct information, keeping track of what's being typed, and handling form submission. **Formik is a library** that helps make this process much easier.

## Why do we need Formik for forms?

1. **Simplified State Management**: Formik automatically tracks the values entered in the form fields. Without it, you'd need to write a lot of code to handle state for each input field manually.
2. **Built-in Validation**: Formik helps you quickly add validation rules (like checking if an email is valid or a password is strong). This saves you from having to write custom validation code.
3. **Easy Form Submission**: Formik takes care of form submission, so you don't have to manually collect and send the data. It also handles things like preventing the form from submitting if the data is invalid.
4. **Less Boilerplate Code**: Without Formik, you'd need to write a lot of repetitive code for managing form state, handling validation, and submitting the form. Formik does all this in a clean and simple way, reducing the amount of code you need to write.
5. **Error Handling**: Formik makes it easier to display error messages when something goes wrong with the form (like a missing required field).

In short, Formik saves you time and effort by handling the complex parts of working with forms in React.

## What is Formik?

Formik is a library for React that makes handling forms easier. When you create a form in React, you need to manage things like:

- What the user types into the form.

- Checking if the input is correct (like making sure the email is valid).
- Sending the data somewhere (like a server).

## useFormik Hook

The **useFormik** hook is a special function provided by Formik to help you manage form data in React. It gives you an easy way to work with form values, handle validation, and handle form submission all in one place.

### How does it work?

**Initialization**: When you use **useFormik**, you give it an object to set up the form. This object can include:

- **initialValues**: The starting values for each input field (like empty strings or default values).
- **onSubmit**: A function that runs when the form is submitted, typically to handle sending data to a server or doing something with the form data.

**Tracking Values**: **useFormik** automatically tracks the values entered in your form fields. You can access these values through **formik.values**. This makes it easy to see what the user has entered at any time.

**Handling Changes**: Whenever a user types something into an input field, **useFormik** provides a **handleChange** function. You use this function to update the form values. It automatically updates the corresponding value in **formik.values** based on the field name.

**Form Submission**: When the form is submitted (for example, when the user clicks a submit button), the **handleSubmit** function provided by **useFormik** is called. This function runs your **onSubmit** function, passing all the current form values as arguments.

### Working steps

#### 1. Install Formik

First, you need to install Formik in your React project.

```javascript
npm install formik
```

### 2. Import the useFormik Hook

In your React component, import the useFormik hook from Formik:

```javascript
import { useFormik } from 'formik';
```

### 3. Set Up the useFormik Hook

Call the useFormik hook in your component to initialize your form. Provide three main things:

- **initialValues**: The starting values of your form fields.
- **onSubmit**: A function that runs when the form is submitted.
- **Optional: validate**: A function for custom validation of form data.

```javascript
const formik = useFormik({
  initialValues: {
    name: '',   // Initial value for "name"
    email: '',  // Initial value for "email"
  },
  onSubmit: (values) => {
    console.log(values); // This runs when the form is submitted
  },
  validate: (values) => {
    const errors = {};
    if (!values.name) {
      errors.name = 'Name is required';
    }
    if (!values.email) {
```

```
      errors.email = 'Email is required';
    }
    return errors; // Return an object with error messages
  },
});
```

### 4. Create a Form

JavaScript

```
<form onSubmit={formik.handleSubmit}>
  {/* Input for Name */}
  <input
    type="text"
    name="name"
    value={formik.values.name}       // Tracks the value for "name"
    onChange={formik.handleChange}   // Updates the value when user types
    onBlur={formik.handleBlur}       // Optional: Marks field as "touched"
  />
  {formik.errors.name && <div>{formik.errors.name}</div>}  {/* Show error if exists */}

  {/* Input for Email */}
  <input
    type="email"
    name="email"
```

```
    value={formik.values.email}      // Tracks the value for "email"
    onChange={formik.handleChange}   // Updates the value when user types
    onBlur={formik.handleBlur}       // Optional: Marks field as "touched"
  />
  {formik.errors.email && <div>{formik.errors.email}</div>} {/* Show error if exists */}

  {/* Submit Button */}
  <button type="submit">Submit</button>
</form>
```

## Handle Validation

Formik automatically checks the fields for changes and applies the validation rules you define in the validate function. Errors are shown using **formik.errors**.

## Submit the Form

When the user clicks the "Submit" button:

- Formik automatically runs your **onSubmit** function.
- It passes all the form values as an object to the function.

Let's understand with simple real-life example

## Registration Form

**Problem**: You want to create a form to register users. The form asks for:

- Name, email, and password.

- Confirm password (should match the password field).

**Formik Solution**:

- Use **useFormik** to handle all four fields.
- Validate that:
  - All fields are required.
  - Password and confirm password match.

Solution:
GitHub_Link

Code Explanation
GitHub_Link

Deployment_Link

Link

## useDebounce Hook

**B**efore understanding the use of the useDebounce hook. Lets understand ,

### What is Debouncing?

Debouncing in JavaScript is a way to **control how often a function runs**. It makes sure that the function only runs after a certain amount of time has passed since the last time it was triggered.

Let's try to understand some simple real-life scenarios

### Ordering Food at a Restaurant

Suppose you're at a restaurant, and the waiter is asking for your order. You're unsure and keep changing your mind: "I'll have pizza... no, pasta... wait, maybe a burger."
The waiter decides to wait for a few seconds after you stop talking to take your final order.

- **Debouncing:** The waiter waits until you're done before acting.

### How it works in real life:

Debouncing is used in scenarios like:

- **Search boxes:** When you type in a search bar, you don't want to search after every keystroke. Instead, wait for the user to stop typing for a moment before showing search results.
- **Resizing windows:** When resizing a window, debounce prevents constantly firing a function and waits until resizing stops.

### Advantages of Debouncing

#### 1. Reduces Unnecessary Function Calls

- Without debouncing, functions (like API calls, DOM updates, or event handlers) might execute too frequently, leading to performance bottlenecks.

- Debouncing ensures the function executes only **after a pause** in the event, reducing the frequency of execution and preventing redundant operations.

**Example:** In a search bar, instead of making an API call for every keystroke, debouncing ensures the call happens only after the user finishes typing.

### 2. Improves Application Performance

- Frequent function executions (e.g., API calls or DOM re-renders) can slow down the app and degrade the user experience.
- Debouncing minimizes these executions, keeping the app responsive and fast.

### 3. Optimizes Network Usage

- Without debouncing, a user typing quickly into a search bar could trigger **dozens of API requests** in a few seconds.
- Debouncing prevents this by sending just **one API request** after the user stops typing, reducing network load and saving bandwidth.

Now, let's come to points to implement such functionality developers are creating a custom hook called **useDebounce.**

### What is useDebounce hook?

**useDebounce** is not a built-in React hook. It's typically a **custom hook** that developers create to add debouncing functionality to React applications. Debouncing is a technique used to limit the rate at which a function is executed, especially for events like typing, resizing, or scrolling.

Lets understand it through simple real-life example

**Problem**:You have a website or app that adjusts its layout (like font size, column count, or visibility of elements) when the window is resized. But if you update the layout every time the window is resized,

it can slow down the performance because resizing happens continuously. You only want the layout to update once the user has finished resizing the window, not while they are still resizing.

Hints - To avoid this, you can use a technique called **debouncing**. This means you wait until the user stops resizing the window for a short period (e.g., 300ms) before making any changes to the layout. This prevents unnecessary updates while the user is actively resizing the window.

**Solution:**

GitHub_Link

Code Explanation

GitHub_Link

Deployment_Link

Link

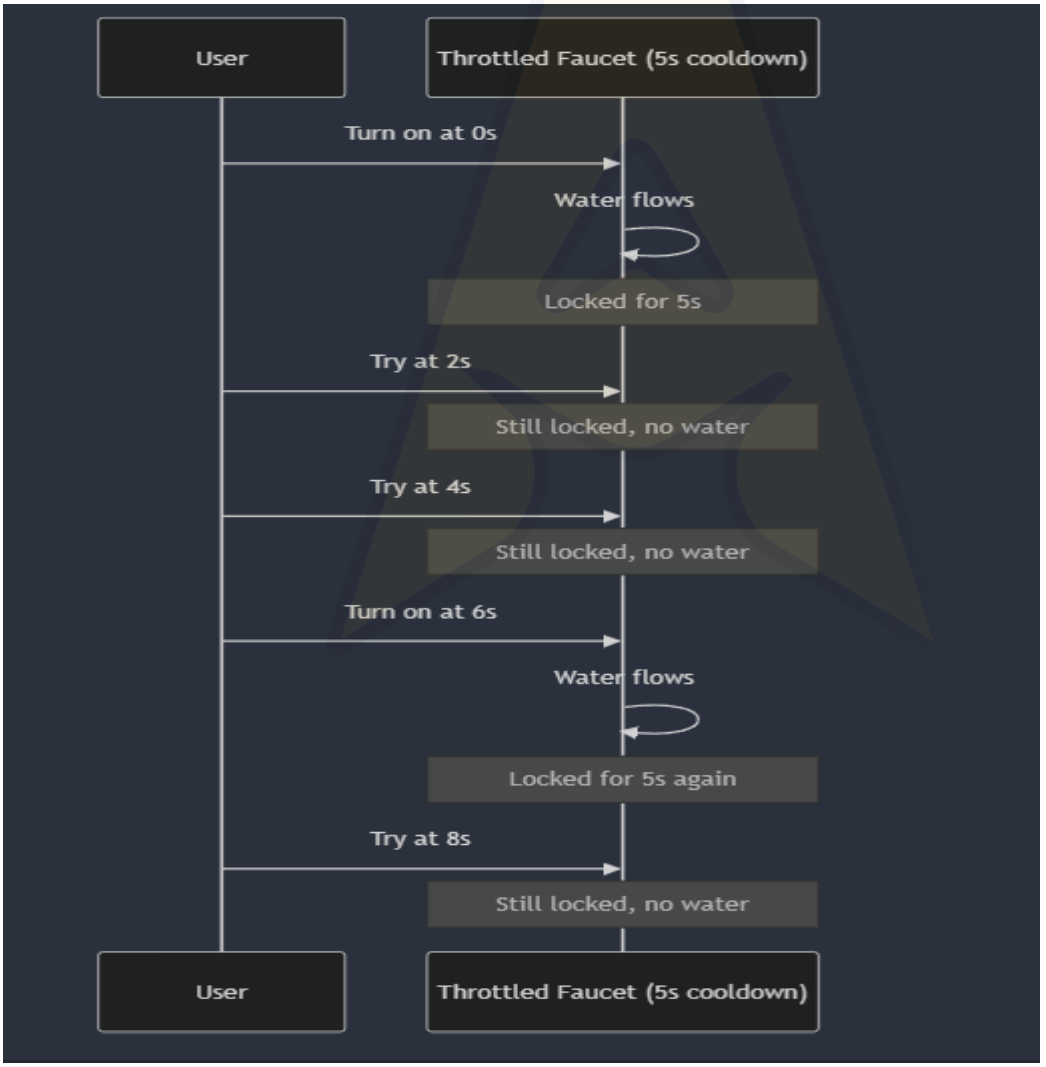
## useThrottle Hook

**B**efore going to useThrottle hook lets understand,

### What is throttling?

**Throttling** is a technique that **limits how often a function can be called in a given period of time.** It is useful for improving the **performance and responsiveness** of web pages that have event listeners that trigger heavy or expensive operations, such as animations, scrolling, resizing, fetching data, etc.

Sometimes it is more complicated to understand . Let's understand it more simple way




Let's understand how throttle works through this digram

Here in this diagram throttling by comparing it to a **faucet (tap)** that has a **5-second cooldown period**.

### What's Happening Here:

1. **Turn on at 0 seconds:**
   - The user opens the faucet.
   - Water flows immediately.
   - After that, the faucet **locks itself for 5 seconds** (cooldown starts).
2. **Try again at 2 seconds:**
   - The user tries to open the faucet again.
   - **Nothing happens**, because the faucet is still in its 5-second cooldown.
3. **Try again at 4 seconds:**
   - The user tries again.
   - **Still nothing happens**, because the cooldown isn't finished yet.
4. **Turn on at 6 seconds:**
   - The cooldown is over.
   - The faucet works again, and water flows.
   - The cooldown resets for another 5 seconds.
5. **Try again at 8 seconds:**
   - The user tries again before the cooldown ends.
   - No water flows, because the faucet is still locked.

### How This Relates to Throttling in JavaScript:

- The **faucet** is like a **throttled function**.
- The **5-second cooldown** is the time limit set for the throttling.
- Even if you try to run the function multiple times during the cooldown, it won't work until the cooldown is over.

### Advantages of throttling

### Improves performance:

- Events like scrolling, resizing the window, or mouse movement can trigger thousands of actions in a second. Throttling reduces

how often your function runs, so your app doesn't waste resources handling unnecessary calls.

**Example:** When scrolling, you might want to load new content or update a progress bar. Without throttling, your app will try to do this continuously, which can lag the page.

**Prevents crashes:**

- If you run a function too many times, it can overwhelm the browser or server, leading to crashes or freezing.
- **Example:** Clicking a "submit" button too fast could accidentally send hundreds of requests to a server.

**Saves battery and energy:**

- For mobile apps or websites running on phones, throttling helps save battery life by reducing the number of unnecessary function calls.

**Ensures consistent behavior:**

- Throttling ensures your function runs at predictable intervals, making it easier to manage animations, UI updates, or data fetching.

Now, let's come to points to implement such functionality developers are creating a custom hook called **useThrottle.**

**What is useThrottle hook?**

The **useThrottle hook** is a custom React hook that helps you **throttle a value or function** in your React app. It's useful when you want to control how often a function or value gets updated, even if the input is changing rapidly.

Lets understand it through simple real-life example

## Button Click Rate Limiting

### Problem:

When a user clicks a "Like" button too fast, multiple requests are sent to the server. We want to limit the number of times the "Like" button can trigger the request, allowing it to happen only once every 2 seconds.

Hints - We'll create a **useThrottle** hook that limits the frequency of function calls (like the "Like" button click handler) to once every 2 seconds.

Solution:

GitHub_Link

Code Explanation

GitHub_Link

Deployment_Link

Link

AIMERZ.ai
Aim.Act.Achieve

# Thank You

## Follow Us


Vishwa Mohan


Vishwa Mohan


vishwa.mohan.singh


Vishwa Mohan


Aimerz


Aimerz