

**Full Stack AI Software Development**

**React Hooks**

**Job Connector Program**

# Outline

## Core Concepts of React Hooks

Hooks let functional components manage state, run side effects, and access React features without using classes, making logic cleaner and more reusable.

## Essential Hooks for Real Projects

Key hooks like useState, useEffect, useRef, useMemo, useCallback, useContext, and useReducer handle state, side effects, performance optimization, and shared data management.

## Reusable Logic with Custom Hooks

Custom hooks allow developers to extract, share, and reuse stateful logic across components, leading to cleaner, more maintainable codebases.

# What is React Hooks?

**React hooks** is functions that let functional components store state, run side effects, and tap into React features without using classes.

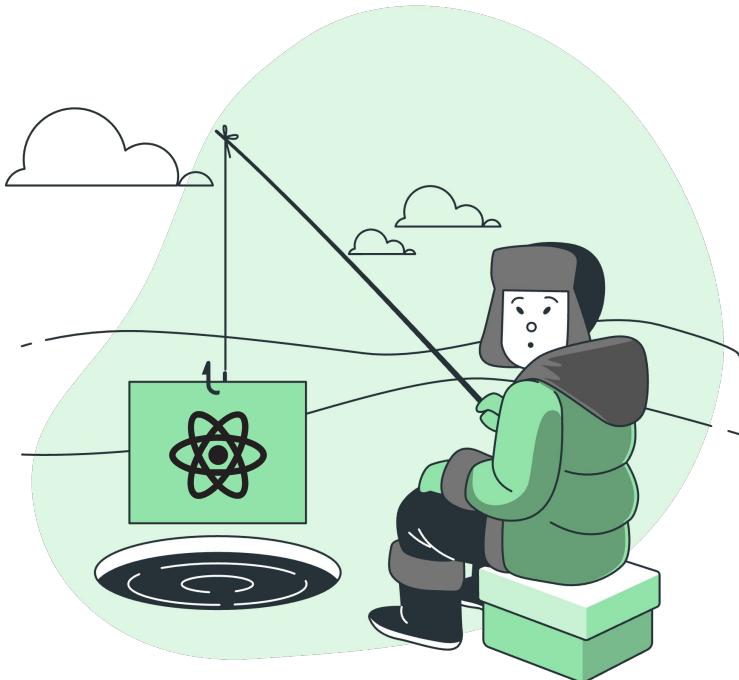
## Why They Matter?

They simplify logic, reduce boilerplate, and make components easier to reuse and test.

## How Hooks Help

Hooks act like plug-ins you attach to a function component:

- **useState** gives the component a memory slot to store values (e.g., a counter).
- **useEffect** schedules actions when something changes (e.g., fetch data on load).
- **useContext** shares one source of truth across many components.
- **useRef** keeps mutable values or DOM references.
- **useMemo / useCallback** prevent unnecessary re-calculations.



# React Hooks Benefits and Purposes in React Applications

## Simplified State Management

It makes state management easy in functional components, letting you create and control state without classes.

## Enhanced Code Reusability

You can build custom hooks to bundle and reuse stateful logic across components, giving your code better reusability and a more modular structure.

## Improved Readability

Functional components with hooks are simpler and clearer than class components, making the code cleaner and easier to maintain.

## Reduced Boilerplate

Hooks eliminate the need for writing constructor methods, using this references, and the render method, resulting in less boilerplate code.

## Easy Side Effect Handling

It streamlines handling side effects like fetching data or updating the DOM, letting you control when effects run and avoid unnecessary re-renders.

## Improved Performance

These hooks optimize components by memoizing values and functions, reducing extra re-renders and boosting performance.

# Most Commonly Used React Hooks

- **useRef**

Stores a mutable value between renders without causing re-renders. Commonly used to access DOM elements directly.

- **useState**

Manages internal component state and triggers re-renders when the value changes.

- **useEffect**

Runs side effects such as data fetching, subscriptions, or DOM updates. Can be controlled with dependency arrays.

- **useMemo**

Memoizes expensive calculations to avoid unnecessary recomputations.

- **useContext**

Provides access to shared/global state across the component tree without prop drilling.

- **useReducer**

Handles state with more complex logic or multiple transitions, similar to Redux-style reducers.

- **useCallback**

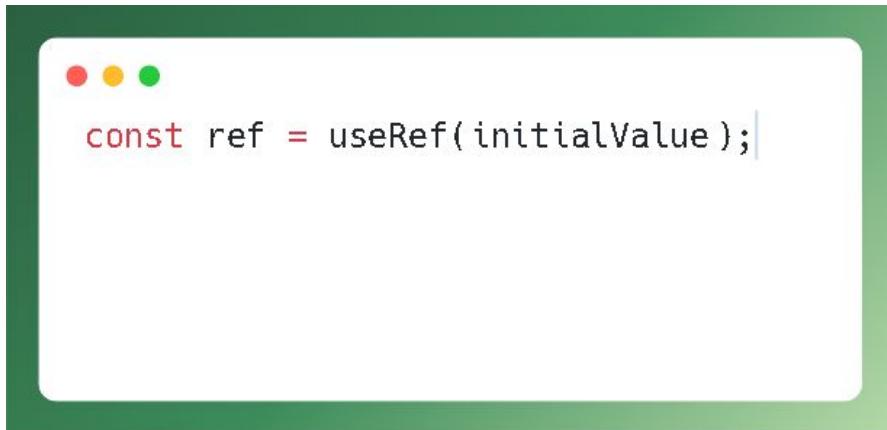
Memoizes function definitions to prevent unnecessary re-creation on re-renders.

**Reference**

<https://react.dev/reference/react/hooks>

# useRef

**useRef** creates a mutable object that can store a DOM reference or any value without causing a re-render. Unlike state, changes to a ref do not trigger the component to render again.

A screenshot of a code editor window. At the top left are three colored circular icons: red, yellow, and green. Below them, the code `const ref = useRef(initialValue);` is written in a monospaced font. The code editor has a dark background with a light gray sidebar on the left and a white main area for the code. A vertical scroll bar is visible on the right side of the editor window.

```
const ref = useRef(initialValue);
```

**ref** : an object with a single property **.current**

**initialValue** : the starting value of the reference

The **.current** property can be read or updated freely, and React won't re-render the component when it changes.

# useRef

```
● ● ●  
import { useRef } from "react";  
  
function FormPage() {  
  const inputRef = useRef<HTMLInputElement>(null);  
  
  const handleGetValue = () => {  
    const value = inputRef.current.value;  
    alert("Input Value: " + value);  
  };  
  
  return (  
    <div>  
      <input  
        ref={inputRef}  
        placeholder="Type something..."  
      />  
      <button onClick={handleGetValue}>  
        Get Value  
      </button>  
    </div>  
  );  
}  
  
export default FormPage;
```

A common use case is reading the value of an input field without using state. Instead of updating state on every keystroke, you simply read the value from **inputRef.current.value** when you need it. This is useful when you only want the value at certain moments — for example, when the user clicks a button.

## How It Works

- **useRef(null)** creates a container that will hold the input element.
- **ref={inputRef}** tells React to store the real `<input>` element into **inputRef.current**.
- When the button is clicked, we read the actual text typed by the user through **inputRef.current.value**.
- No re-renders happen because changing **.current does not update the UI**.

# useState

**useState** is one of the most important and commonly used React Hooks. It lets you add and manage state inside a function component. State is data that can change over time and cause the component to re-render when it updates.

Example use of useState:

- **Managing a counter value** — for example, to increase or decrease the quantity of a product item in the shopping cart.
- **Handling form input fields** — such as storing and updating the value that users type into text input or select menus.
- **Controlling modal visibility** — toggling whether a modal (popup dialog) is open or closed based on user interaction.

# useState

How It Works:

- **useState(0)** creates a state value with an initial value of 0.
- **count** stores the current state value and is used inside the UI.
- **setCount** is the function that updates count and triggers a re-render.
- By writing **useState<number>(0)**, you restrict the state so it can only contain numeric values.
- Every time **setCount** is called, React **re-renders the component** with the **updated value**.

```
● ● ●
import { useState } from 'react';

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

  return (
    <>
      <h2>{count}</h2>
      <button onClick={() => setCount((count) => count + 1)}>Counting</button>
    </>
  );
}

export default App;
```

# useReducer

**useReducer** is like **useState**. But , it will useful if you had **complex logic to manage your state**.

useReducer work with **useDispatch** to **triggering some logic function inside reducer**.

Take a look at this simple code. Looks like we can handle it only with **useState**.

```
●●●  
import { useState } from 'react';  
  
function App() {  
  const [count, setCount] = useState<number>(0);  
  
  const increment = () => {  
    setCount(count + 1);  
  };  
  
  const decrement = () => {  
    setCount(count - 1);  
  };  
  
  return (  
    <div>  
      <div>  
        <button onClick={decrement}>-</button>  
        Count: {count}  
        <button onClick={increment}>+</button>  
      </div>  
    </div>  
  );  
}  
  
export default App;
```

# useReducer

```
import { useReducer } from "react";

// ❶ Define types
type State = {
  count: number;
};

type Action = { type: "increment" } | { type: "decrement" };

// ❷ Define reducer function
function reducer(state: State, action: Action): State {
  switch (action.type) {
    case "increment":
      return { count: state.count + 1 };
    case "decrement":
      return { count: state.count - 1 };
    default:
      return state;
  }
}

// ❸ Initial state
const initialState: State = { count: 0 };

// ❹ Component
function App() {
  const [state, dispatch] = useReducer(reducer, initialState);

  return (
    <div style={{ textAlign: "center", marginTop: "2rem" }}>
      <h1>Count: {state.count}</h1>
      <button onClick={() => dispatch({ type: "decrement" })}>-
        </button>
      <button onClick={() => dispatch({ type: "increment" })}>+
        </button>
    </div>
  );
}

export default App;
```

- With **useReducer**, we can group our state and the functions that update it in **one place**.
- We trigger changes by dispatching actions using **dispatch()**.
- This pattern is helpful because it keeps our logic **organized** and **easier to manage** as the app grows.
- Understanding **useReducer** will also make it easier to **learn state-management** tools like **Redux Toolkit**, since they use a **similar concept**.

# Handling Side Effects and Cleanup

In React, a *side effect* is anything that happens outside the normal UI rendering process.

Common examples:

- Fetching data from an API
- Updating the page title
- Adding event listeners (scroll, resize)
- Using timers (setTimeout, setInterval)
- Accessing localStorage

React needs a safe way to handle these so that:

- They don't run too many times
- They don't slow down the app
- They don't cause memory leaks

This is why React provides `useEffect`.



# useEffect

**useEffect** is a React Hook that allows you to perform side effects in functional components, such as data fetching, DOM manipulation, etc. It runs code after the component has rendered and accepts a function with the effect's logic and an optional dependency array, which controls when the effect re-runs.

## How useEffect works?

- **Runs after render:** useEffect runs your code after React has updated the DOM, preventing performance issues by not blocking the rendering process.
- **Handles side effects:** It's used for any operations that aren't directly related to rendering, like making an API call when a component mounts or updating the document's title.
- **Uses a dependency array:** This second argument tells React when to re-run the effect.
  - [ ] (empty array): The effect runs only once after the initial render, similar to componentDidMount.
  - [variable1, variable2]: The effect runs after the initial render and any time the value of variable1 or variable2 changes.
  - **No array: The effect runs after every render, which can lead to infinite loops if not managed carefully.**

# useEffect

```
import { useEffect } from 'react';

function App() {
  useEffect(() => {
    | console.log('useEffect Triggered')
  });

  return <>...</>;
}

export default App;
```

## How It Works:

- **No dependency array** means the effect runs after every render.
- React **executes** the function inside useEffect **each time the component updates**.
- In this example, the console message **appears on every render cycle**.
- This pattern is useful when you want to track or respond to all component updates, not just the initial one.

# useEffect

```
import { useEffect } from 'react';

function App() {
  useEffect(() => {
    console.log('useEffect Triggered')
  }, []); // ✅ This empty array makes it run only once on the first render
  return <>...</>;
}

export default App;
```

## How It Works

- An **empty dependency array (`[]`)** means the effect **runs only once**, after the **component's first render**.
- React calls the function inside `useEffect` right **after the component mounts**.
- In this example, the console logs "useEffect Triggered" appears only **one time in the console**.
- This pattern is useful for **one-time actions** like **fetching data, initializing value**, etc.

# useEffect

```
import { useEffect } from 'react';

function App() {
  useEffect(() => {
    // This function runs after the component renders and when props
    // state have changed
    console.log('useEffect Triggered')
  }, [props, state]);
}

return <>...</>;
}

export default App;
```

## How It Works

- The effect **runs after the component first renders**.
- It also **runs again whenever any value in the dependency list changes**.
- When those values update, the effect runs and logs "useEffect Triggered".
- This is useful when you want something to happen only when certain data changes — like getting new data or updating part of the UI.

# Cleanup Function — Cleaning Up Side Effects

Some side effects need to be cleaned up to prevent:

- Memory leaks
- Duplicate event listeners
- Timers running after a component is removed

React handles cleanup using a **return function inside useEffect**.

General pattern:

```
useEffect(() => {
  // setup code

  return () => {
    // cleanup code
  };
}, []);|
```



# Example Cleanup

Without proper cleanup:

- Timers continue running in the background
- Event listeners keep piling up
- Performance drops
- “Memory leak” warnings appear
- Components try to update state after they are unmounted

```
● ● ●  
// Removing an Event Listener  
useEffect(() => {  
  function handleScroll() {  
    console.log("Scrolling...");  
  }  
  
  window.addEventListener("scroll", handleScroll);  
  
  return () => {  
    // cleanup  
    window.removeEventListener("scroll", handleScroll);  
  };  
, []);  
  
// Clearing an Interval  
useEffect(() => {  
  const id = setInterval(() => {  
    console.log("Tick...");  
  }, 1000);  
  
  return () => {  
    clearInterval(id); // stop the interval when component unmounts  
  };  
, []);|
```

# useMemo

**useMemo** is a React Hook that helps you optimize performance by memoizing (caching) the result of a computation — so React doesn't recompute it unnecessarily on every render.

Use `useMemo` when:

- You have expensive calculations (e.g., filtering, sorting, heavy loops).
- You want to avoid re-calculating something unless its inputs change.
- You need to prevent unnecessary re-renders caused by derived values.



```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

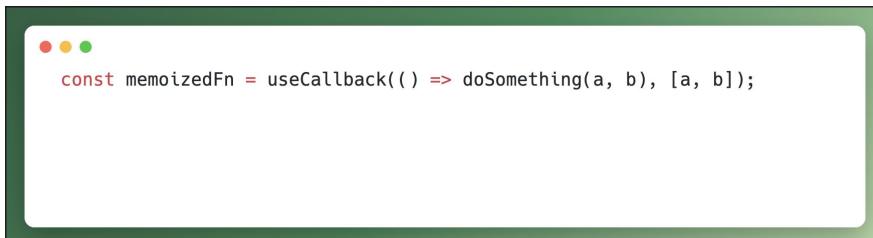
- The **first argument** is a function that returns the computed value.
- The **second argument** is a dependency array — when any value inside it changes, React will re-run the function.
- If the dependencies don't change, React will **reuse (or “memoize”)** the previous result.

# useCallback

**useCallback** is a React Hook that returns a memoized version of a function, meaning React will only re-create the function when its dependencies change. This is useful when passing functions to child components, especially when those children are wrapped with **React.memo**, because a new function reference normally triggers re-renders.

## Use useCallback when:

- You pass a function as a prop to a child component
- You want to prevent unnecessary re-renders caused by changing function references
- You need a stable function identity for optimization (e.g., with `React.memo`, `useEffect`, or `useImperativeHandle`)
- The logic inside the function doesn't need to change on every render



```
const memoizedFn = useCallback(() => doSomething(a, b), [a, b]);
```

- **The first argument** is the function whose reference you want to stabilize.
- **The second argument** is a dependency array — when any value inside it changes, React will generate a new version of the function.
- If the dependencies remain the same, React will **reuse the previous function reference** instead of creating a new one.

# useContext

**useContext** is a React Hook that helps you **share data between components without passing props through every level.**

Normally, if Component 1 needs to send data to Component 3, you must pass props through Component 2 — even if Component 2 doesn't use them.

With useContext, you can skip this “**prop drilling**” and let Component 3 **get the data directly**.

```
● ● ●  
import { useState } from 'react';  
  
function Component1() {  
  const [user, setUser] = useState<string>('Rizky Ridho');  
  
  return (  
    <>  
      <h1>Hello {user}!</h1>  
      <Component2 user={user} />  
    </>  
  );  
}  
  
function Component2({ user }: { user: string }) {  
  return (  
    <>  
      <h1>Component-02</h1>  
      <Component3 user={user} />  
    </>  
  );  
}  
  
function Component3({ user }: { user: string }) {  
  return (  
    <>  
      <h1>Component-03</h1>  
      <h2>{'Hello again, ${user}!'</h2>  
    </>  
  );  
}
```

# useContext

```
•••  
import { useState, useContext } from 'react';  
  
const UserContext = createContext<string>'');  
  
function Component1() {  
  const [user, setUser] = useState<string>('Rizky Ridho');  
  
  return (  
    <>  
      <UserContext.Provider value={user}>  
        <h1>Hello ${user}!</h1>  
        <Component2 />  
      </UserContext.Provider>  
    </>  
  );  
}  
  
function Component2() {  
  return (  
    <>  
      <h1>Component 2</h1>  
      <Component3 />  
    </>  
  );  
}  
  
function Component3() {  
  const user = useContext<string>(UserContext);  
  return (  
    <>  
      <h1>Component 3</h1>  
      <div>Hello {user} again</div>  
    </>  
  );  
}
```

- Use **createContext** to create a **shared data container**.
- **Wrap the components that need the data** with this **Context Provider**.
- **Any component** inside the provider **can access the value directly**.
- Use **useContext** to **read the shared value** without **passing props** through **every component layer**.

# Newest React Hooks from ReactJS

- **useId**  
Generates unique IDs for accessibility and server-side rendering (SSR) purposes. This ensures there are no mismatches during hydration.
- **useDeferredValue**  
Defers updating a value until higher-priority updates are completed, improving performance during rendering.
- **useTransition**  
Allows you to prioritize urgent updates while deferring less critical ones, ensuring smooth transitions.
- **useSyncExternalStore**  
For subscribing to external data sources while maintaining compatibility with concurrent rendering.
- **useInsertionEffect**  
Used for injecting styles or effects directly before DOM mutations for better rendering efficiency.
- **useFormStatus**  
Tracks the submission status of a form, allowing conditional UI updates during asynchronous actions.
- **useActionState**  
Monitors the state of actions, simplifying handling of loading, success, or error states in components.

## Reference

<https://www.freecodecamp.org/news/react-19-new-hooks-explained-with-examples/>

# Rules of Hooks

Hooks are JavaScript functions, but you need to follow two rules when using them.

## Only Call Hooks at the Top Level

- Don't use Hooks inside loops, conditions, or nested functions.
- Always place Hooks at the top of your component, before any early returns.
- This keeps the order of Hook calls consistent on every render, allowing React to correctly maintain state.

## Only Call Hooks from React Functions

- Don't call Hooks inside regular JavaScript functions.
- You may only call Hooks from:
  - React components
  - Custom hook
- This ensures all stateful logic is easy to see and track in your component.

# Custom Hook

Creating your own Hooks lets you move **reusable logic** out of components.

In the example, we build a simple counter using useState that increases or decreases when the buttons are clicked.

If we need the same counter logic in multiple parts of the app, we can put it inside a custom Hook instead of repeating the code.

A custom Hook is just a **JavaScript function that starts with use and can use other React Hooks**.

```
// App.ts
import { useState } from 'react';

function App() {
  const [counter, setCounter] = useState<number>(0);

  const increment = () => {
    setCounter(counter + 1);
  };

  const decrement = () => {
    setCounter(counter - 1);
  };

  return (
    <div>
      <button onClick={decrement}>-</button>
      <h3>{counter}</h3>
      <button onClick={increment}>+</button>
    </div>
  );
}

export default App;
```

# Create your own hook

```
● ● ●  
// useCounter.ts  
import { useState } from 'react';  
  
function useCounter(val: number = 0, step: number = 1): [number, () => void, () => void]  
{  
  const [count, setCount] = useState<number>(val);  
  
  const increment = () => setCount(count + step);  
  const decrement = () => setCount(count - step);  
  
  return [count, increment, decrement];  
}  
  
export default useCounter;  
  
|  
// App.ts  
import useCounter from './useCounter';  
  
function App() {  
  const [count, increment, decrement] = useCounter(0, 1);  
  
  return (  
    <div>  
      <h1>{count}</h1>  
      <button onClick={increment}>increment</button>  
      <button onClick={decrement}>decrement</button>  
    </div>  
  );  
}  
  
export default App;
```

## How it works

- We move the counter logic into a custom Hook called **useCounter**.
- This Hook **accepts two parameters**:
  - **val** → The initial counter value
  - **step** → How much the counter increases or decreases
- The Hook returns:
  - **count** (current value)
  - **increment()** to increase
  - **decrement()** to decrease
- In **App.ts**, we **simply import** and **call useCounter**.
- The counter logic is now **reusable anywhere in the app**.

# Exercise

- Continue exercise day 4, create the add, update and delete features to-do list

# Thank you

