

# FRONTEND-CONCEPTS

## HOOKS

### What is HOOKS ?

Hooks are the functions to use some react features in functional components . in other words, Hooks are functions that make functional components work like class components.

### TYPES OF HOOKS :

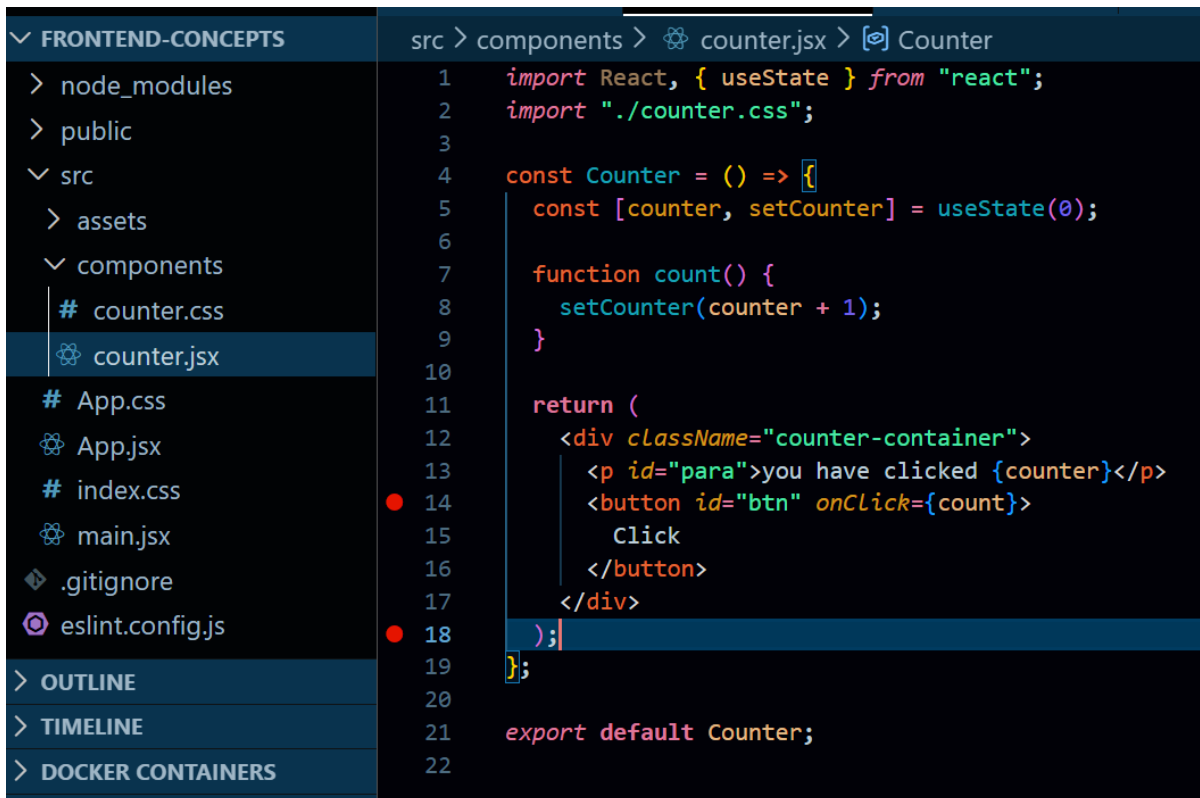
- 1) useState
- 2) useEffect
- 3) useCallback
- 4) useContext
- 5) useRef
- 6) useMemo

1) **useState Hook** : it is a function to add state in functional component.

What is state ?

State is nothing but just values / variables of your component.

useState consists of [state variable , state function]



The screenshot shows a code editor with a file explorer on the left and a code editor on the right. The file explorer shows a project structure with folders like 'node\_modules', 'public', and 'src'. The 'src' folder is expanded, showing files like 'App.css', 'App.jsx', 'index.css', 'main.jsx', '.gitignore', and 'eslint.config.js'. The 'counter.jsx' file is selected. The code editor shows the following code:

```
src > components > counter.jsx > Counter
1  import React, { useState } from "react";
2  import "./counter.css";
3
4  const Counter = () => {
5    const [counter, setCounter] = useState(0);
6
7    function count() {
8      setCounter(counter + 1);
9    }
10
11   return (
12     <div className="counter-container">
13       <p id="para">you have clicked {counter}</p>
14       <button id="btn" onClick={count}>
15         Click
16       </button>
17     </div>
18   );
19 }
20
21 export default Counter;
22
```

## 2) useEffect :

it is a react hook which generates sideEffects.

### What is sideEffect?

Side Effects are the actions which are performed with the “Outside World”

#### Scenarios when to use the useEffect Hook

- Fetching the data from the Api
- Updating the dom document & window
- Timer function SetTimeout & setInterval

A side effect is anything that:

- Fetches data
- Updates DOM manually
- Sets timers
- Adds event listeners
- Uses localStorage
- Calls APIs

#### Basic Syntax

```
useEffect(() => {  
  // side effect code here  
}, [dependencies]);
```

- First argument → function (effect)
- Second argument → dependency array

### Three Types of useEffect :

#### Type 1: No Dependency Array

```
useEffect(() => {  
  console.log("Runs every render");  
});
```

Runs:

- On mount
- On every re-render

Use Case:

- Rarely used
- Mostly for debugging
- Can cause infinite loops if updating state inside it.

#### Type 2: Empty Dependency Array []

```
useEffect(() => {  
  console.log("Runs only once");  
});
```

```
}, []);
```

Runs:

- Only on first render (mount)
- Never again

Use Case:

- Fetch API on load
- Set timer once
- Add event listener once

### **Type 3: With Dependency [value]**

```
useEffect(() => {  
  console.log("Runs when value changes");  
}, [value]);
```

Runs:

- On mount
- When value changes
- Not when other states change

Use Case:

- Reacting to state changes
- API call when id changes
- Form validation

### **Mount vs Re-render vs Unmount**

- o **Mount**  
First time component appears.
- o **Re-render**  
When state or props change.
- o **Unmount**  
When component is removed.

How React Executes useEffect Internally

1. React runs component function.
2. JSX is returned.
3. DOM updates.
4. React checks dependency array.
5. If needed → effect runs.

Dependency Comparison

React does a shallow comparison.

**Example:**

```
useEffect(() => {}, [count]);
```

React compares:

- Old count
- New count

If different → run effect

If same → don't run

### Infinite Loop Example

```
useEffect(() => {  
  setCount(count + 1);  
});
```

Why infinite loop?

1. Effect runs
2. State updates
3. Re-render
4. Effect runs again
5. Loop forever

Fix:

```
useEffect(() => {  
  setCount(count + 1);  
}, []);
```

Cleanup Function

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

```
  return () => {  
    clearInterval(timer);  
  };  
}, []);
```

Cleanup runs:

- Before next effect
- On unmount

Used for:

- Removing event listeners
- Clearing timers
- Cancelling subscriptions

Syntax :

```
2  useEffect(() => {  
3    // Side effect code here  
4  }, [dependencies]);
```

In above code:

- **Effect Function:** The code within the first argument represents the side effect you want to execute.
- **Optional Dependency Array:** The effect re-runs whenever any value in the array changes. If it's empty, it will run only once.

**So, here we have 5 variations of useEffect :**

**Variation 1 : runs on every render , without dependency array**

```
useEffect(() => {  
  alert("hi how are you")  
});
```

**Variation 2 : with empty dependency, that runs on only first render**

```
useEffect(() => {  
  alert("i will render only 1st click ");  
}, []);
```

**Variation 3 : it will run every time when count is updated**

```
useEffect(() => {  
  alert("i will run every time when count is updated");  
}, [count]);
```

**Variation 4 : it will render every time when count/total is updated**

```
useEffect(() => {  
  alert("i will render every time when count /total is updated");  
}, [count, total]);
```

**Variation 5 : the phase in React where a component is removed from the DOM, and cleanup functions inside useEffect are executed.**

```
useEffect(() => {  
  console.log("count is updated");  
  return () => {  
    console.log("count is unmounted from UI");  
  };  
}, [count])
```

Examples :

**1) Used for data fetching:**

USESTATE-USEEFFECT

node\_modules

public

src

assets

components

counter.css

counter.jsx

DataFetcher.jsx

MultiEffectComponent.jsx

ResizeComponent.jsx

TimerComponent.jsx

App.css

App.jsx

index.css

main.jsx

.gitignore

eslint.config.js

index.html

OUTLINE

TIMELINE

DOCKER CONTAINERS

DOCKER IMAGES

AZURE CONTAINER REGISTRY

DOCKER HUB

src > components > DataFetcher.jsx > DataFetcher

You, 18 hours ago | 1 author (You)

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

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

function DataFetcher() {

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

const [loading, setLoading] = useState(true);

useEffect(() => {

fetch("https://jsonplaceholder.typicode.com/posts")

.then((response) => response.json())

.then((data) => {

setData(data);

setLoading(false);

});

}, []);

//it will run only on 1st render

return (

<div>

{loading ? ( //here loading is true

<h1>Loading...</h1>

) : (

<ul>

//this will executes once loading is false

{data.map((priyanka) => (

<li key={priyanka.id}>{priyanka.title}</li>

)})}

</ul>

)}

</div>

);

}

export default DataFetcher;

In this DataFetcher component:

- When the component first loads,
  - data is an empty array []
  - loading is true

So initially, React renders:

</> JavaScript

<h1>Loading...</h1>

</> JavaScript

useEffect(() => {

fetch("https://jsonplaceholder.typicode.com/posts")

.then((response) => response.json())

.then((data) => {

setData(data);

setLoading(false);

});

}, []);

what useEffect does here

The empty dependency array [] means:

Run this effect **only once**, after the first render .

It fetches data from the API.

Once the data is received:

- setData(data) updates the data state.
- setLoading(false) changes loading to false.

When state updates:

- React re-renders the component.
- Now loading is false.
- Instead of showing "Loading...", it renders:

```
<ul>
  {data.map((priyanka) => (
    <li key={priyanka.id}>{priyanka.title}</li>
  ))}
</ul>
```

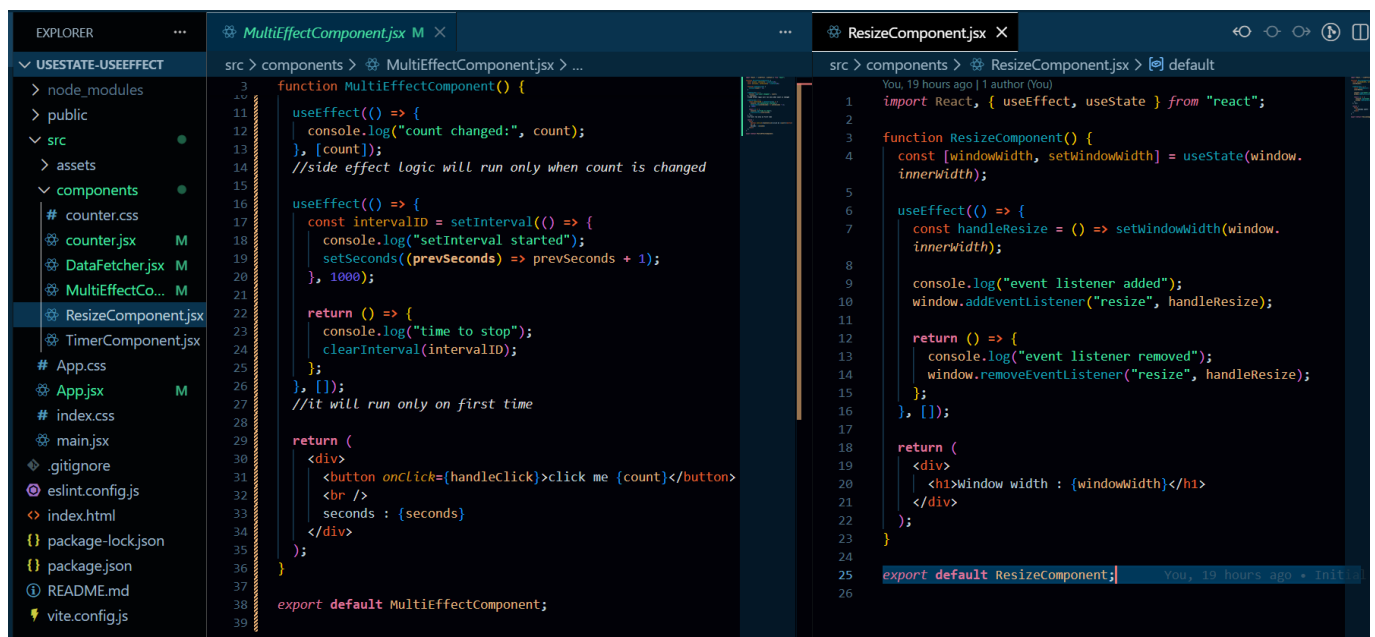
Here:

- data.map() loops through all priyanka.
- Each priyanka is stored temporarily in the variable priyanka.
- It displays each priyanka's title inside a <li>.

Why This Is Efficient

Because of the empty dependency array []:

- The API call happens only once.
- It does not fetch again on every re-render.
- It keeps the app efficient and avoids unnecessary network requests.



### 3) useContext :

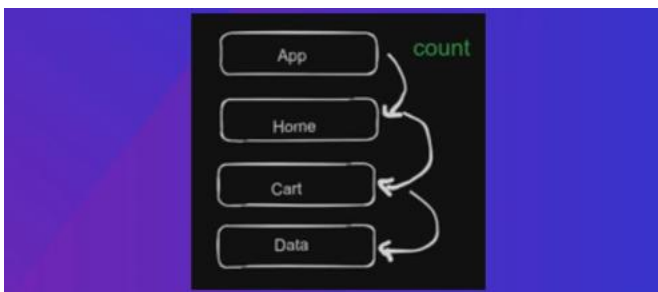
React's Context API is a powerful tool that enables effective state management and facilitates the flow of data through your components without the need for prop drilling.

#### What is Context?

In React, context provides a way to pass data through the component tree without having to pass props down(prop-drilling) manually at every level. It's particularly useful when dealing with global settings, themes, or any information that needs to be accessible by many components.

#### Without context api example:

We are building a simple counter app that passes the count through nested components. The component structure may look like this:



we need to pass data from the parent component (App.jsx) to the grandchild component (Data.jsx). This process is known as prop drilling. However, as our application grows larger in the future, it may pose problems to continually pass data. This is where the Context API comes into play.

#### With Context Api Example:

step 1 : create context outside the function

step 2 : wrap all the child inside a provider

step 3 : pass values

step 4 : go inside the consumer and consume the data

```
App.jsx
1 import React, { createContext, useState } from "react";
2 import "./App.css";
3 import ChildA from "../components/ChildA";
4 //step 1 : create context outside the function
5 //step 2 : wrap all the child inside a provider
6 //step 3 : pass values
7 //step 4 : go inside the consumer and consume the data
8
9 const ThemeContext = createContext();
10 function App() {
11   const [theme, setTheme] = useState("light");
12
13   return (
14     <ThemeContext.Provider value={{ theme, setTheme }}>
15       <div
16         id="container"
17         style={{ backgroundColor: theme === "light" ? "beige" :
18           "black" }}
19       >
20         <ChildA />
21       </div>
22     </ThemeContext.Provider>
23   );
24 }
25 export default App;
26 export { ThemeContext };

ChildC.jsx
1 import React, { useContext } from "react";
2 import { ThemeContext } from "../App";
3
4 const ChildC = () => {
5   const { theme, setTheme } = useContext
6     (ThemeContext);
7
8   function handleTheme() {
9     if (theme === "light") setTheme("dark");
10    else setTheme("light");
11  }
12
13  return (
14    <div>
15      <button onClick={handleTheme}> change
16        theme</button>
17    </div>
18  );
19 }
20 export default ChildC;
```

```
1 import React from "react";
2 import ChildC from "../ChildC";
3
4 const ChildB = () => {
5   return (
6     <div>
7       <ChildC></ChildC>
8     </div>
9   );
10 };
11
12 export default ChildB;
```

This project demonstrates how to use React Context to share data between components without passing props manually at every level.

#### Create Context (App.jsx)

- A ThemeContext is created outside the App function.
- This context will act like a global storage for the theme.

#### Create State (App.jsx)

- A state variable called theme is created.
- Default value is "light".
- A function setTheme is used to update it.

So now:

- theme → holds current theme
- setTheme → updates the theme

#### Wrap Components with Provider (App.jsx)

- All child components are wrapped inside ThemeContext.Provider.
- The Provider shares:

theme  
setTheme

Now every child inside it can access this data.

#### Component Hierarchy (Flow of Components)

App



ChildA



ChildB



ChildC

ChildC is deeply nested, but it can still access the theme directly using Context.

### Consume Context (ChildC.jsx)

- ChildC uses useContext to get:
  - theme
  - setTheme
- No props are passed manually from App → ChildA → ChildB → ChildC.
- This avoids prop drilling.

### Button Click in ChildC

- There is a button inside ChildC.
- When clicked:
  - If theme is "light" → change to "dark"
  - If theme is "dark" → change to "light"

This updates the state in App.

### What Happens After Update?

When theme changes:

- React re-renders App.
- Background color updates.
- All components using theme get the new value automatically.