useEffect Hook

- The useEffect hook is used to handle side effects in functional components.
- Side effects include data fetching, subscriptions, or manually changing the DOM.

Syntax

```
useEffect(() => {

}, [dependency1, dependency2]);
```

- The **useEffect** hook takes two arguments: a **callback** function and an optional **array of dependencies**.
- The callback function represents the side effect you want to perform.
- The dependencies array is used to control when the side effect runs. If any value in the array changes, the side effect is triggered.

Typical Use Cases

- **Data Fetching**: Fetching data from APIs and updating the state with the fetched data.
- **Subscriptions**: Setting up and cleaning up subscriptions to events or services.
- **DOM Manipulation**: Changing the DOM manually when necessary.
- **Timers and Intervals**: Managing timers and intervals in your component.

The Cleanup Function

- The callback function returned from useEffect can be used to perform cleanup operations.
- This is especially useful for unsubscribing from event listeners or clearing up resources to avoid memory leaks.

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

Lets code

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

function EffectTutorial() {
  const [data, setData] = useState("");
  const [count, setCount] = useState(0);

  useEffect(() => {
    axios
      .get("https://jsonplaceholder.typicode.com/co
mments")
      .then((response) => {
        setData(response.data[0].email);
        console.log("API WAS CALLED");
      });
  }, []);
```

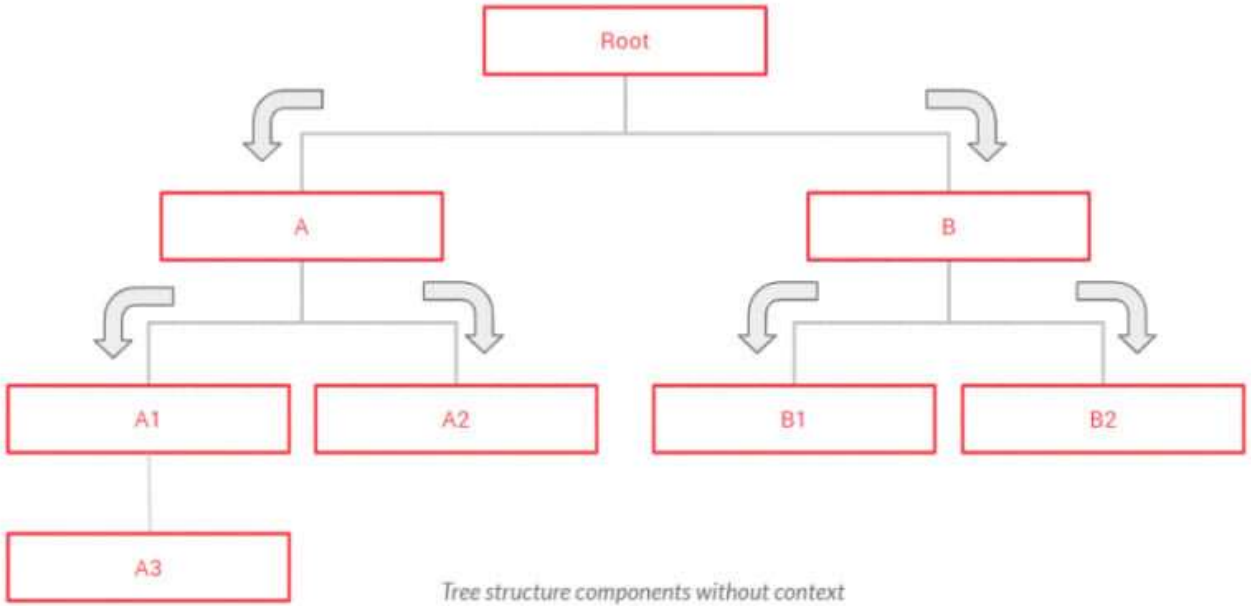
```
  return (
    <div>
      Hello World
      <h1>{data}</h1>
      <h1>{count}</h1>
      <button
        onClick={() => {
          setCount(count + 1);
        }}
      >
        Click
      </button>
    </div>
  );
}

export default EffectTutorial;
```

useContext Hook

- The 'useContext' hook is used to consume data from a React context.
- Context provides a way to share data across the component tree without manually passing props down through each level.
- React Context is a way to manage state globally.
- State should be held by the highest parent component in the stack that requires access to the state

Props Drilling



Tree structure components without context

Setting Up a Context

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

const MyContext = createContext();
export default MyContext;
```

Providing Data with the Context Provider

```
import React from 'react';
import MyContext from './MyContext';

const App = () => {
  const sharedData = { username: ‘galaxe@123', theme: 'dark' };

  return (
    <MyContext.Provider value={sharedData}>
      {/* Your component tree goes here */}
    </MyContext.Provider>
  );
};

export default App;
```

Consuming Data with useContext

```
import React, { useContext } from 'react';
import MyContext from './MyContext';

const MyComponent = () => {
  const sharedData = useContext(MyContext);

  return (
    <div>
      <h1>Welcome, {sharedData.username}!</h1>
      <p>Current theme: {sharedData.theme}</p>
    </div>
  );
};
```

Benefits of useContext

- Avoids prop drilling and keeps your code cleaner.
- Simplifies state management for shared data.
- Makes it easy to access global state within any component.

When to use UseContext

- When you have data that needs to be shared across multiple components without passing it explicitly through props
- When you want to avoid the complexity of prop drilling

useRef Hook

- useRef is a built-in React Hook that creates a mutable object called a "ref."
- Unlike state or props, refs persist across renders and do not trigger re-renders when updated.

Syntax

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

Use Cases of useRef

- Referencing DOM Elements: Access and manipulate DOM elements imperatively without triggering re-renders.
- Managing Previous Values: Keep track of previous values of props or state without using state variables.
- Caching Expensive Computations: Cache the results of expensive computations to avoid recomputing on each render.

```
import React, { useRef, useEffect } from
'react';

const TextInput = () => {
  const inputRef = useRef();

  useEffect(() => {
    inputRef.current.focus();
  }, []);

  return (
    <div>
      <input type="text" ref={inputRef} />
      <button onClick={() =>
inputRef.current.focus()}>Focus
Input</button>
    </div>
  );
};
```