Frontend Development

useReducer Hook:

The useReducer hook in React is a more powerful alternative to useState when dealing with complex state logic in your components. It accepts a reducer function and an initial state, and returns the current state and a dispatch function to update the state based on actions dispatched to the reducer.

The useReducer hook is a function in React that is used for state management. It is an alternative to the useState hook when the state logic is more complex and involves multiple sub-values or when the next state depends on the previous one. The useReducer hook is inspired by the reducer functions in JavaScript, which are commonly used with arrays to reduce them to a single value.

const [state, dispatch] = useReducer(reducer, initialArgument, init);

state: The Current State Value.

dispatch: A Function That Allows You To Dispatch Actions To Modify The State.

Reducer: A Function That Takes The Current State & An Action As Arguments & Returns The New State.

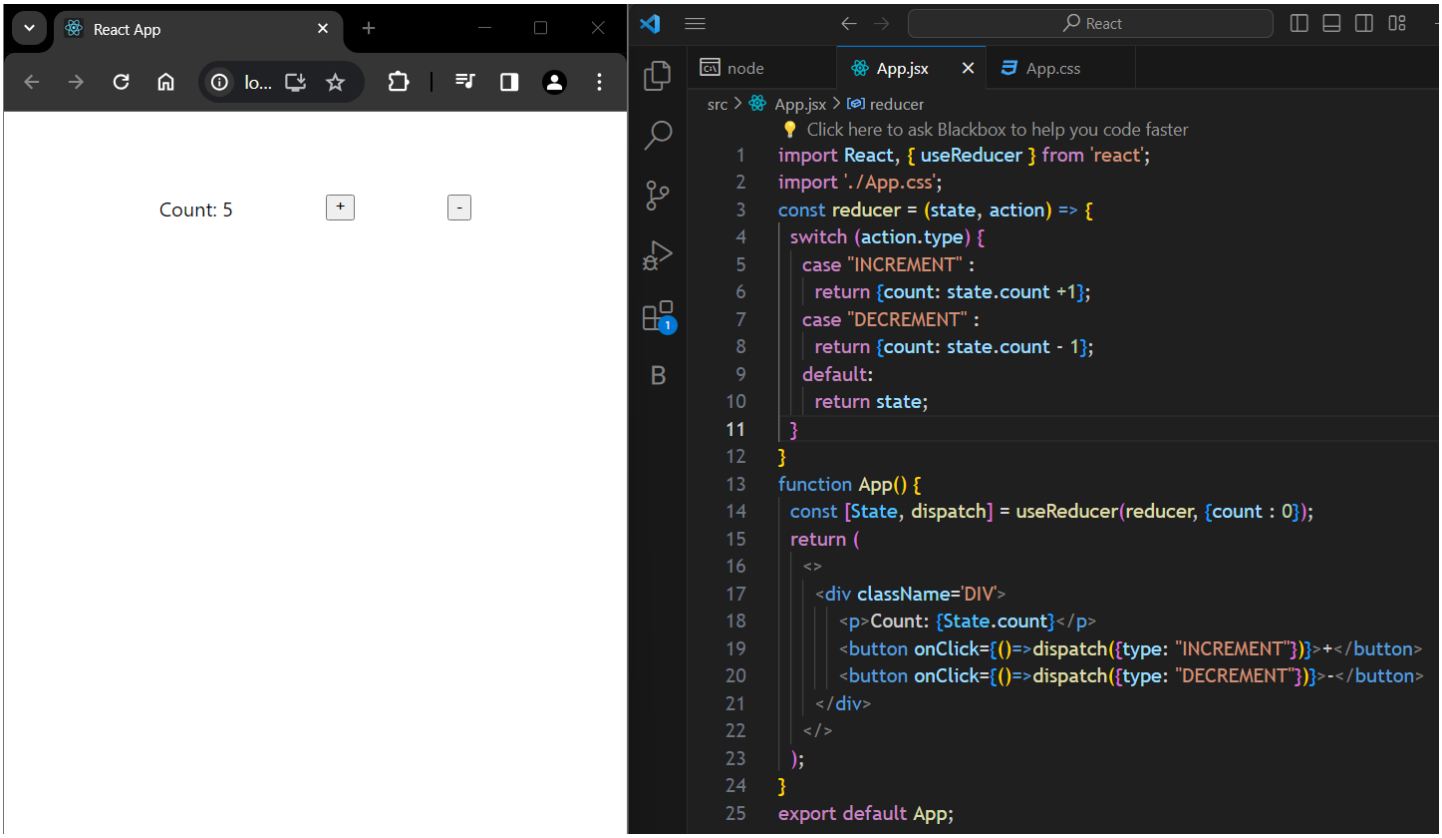
initialArgument: The Initial State Value Or A Function That Returns The Initial State. The Argument Is Optional.

Init: A Function That Returns The Initial State. This Argument Is Optional.

How useReducer Function Works:

useReducer takes a reducer function and an initial state as arguments. When the dispatch function is called with an action, useReducer passes the current state and the action to the reducer function.

The reducer function then calculates the new state based on the current state and the action. useReducer returns the new state, which can be accessed through the state variable, and a dispatch function that can be used to update the state.



1. We define a reducer function that takes the current state and an action, then returns the new state based on the action.

2. In the Counter component, we use useReducer with the reducer function and an initial state { count: 0 }.

3. We use the dispatch function to increment or decrement the count value in the state.

Why useReducer in this above example:

1. We use useReducer because the state logic involves incrementing and decrementing a count value, which is a bit more complex than a simple state update.

2. The reducer function helps manage the state changes in a more structured way compared to handling the logic directly in the component.

3. useReducer allows us to separate the state management logic from the component rendering logic, making the code easier to understand and maintain.

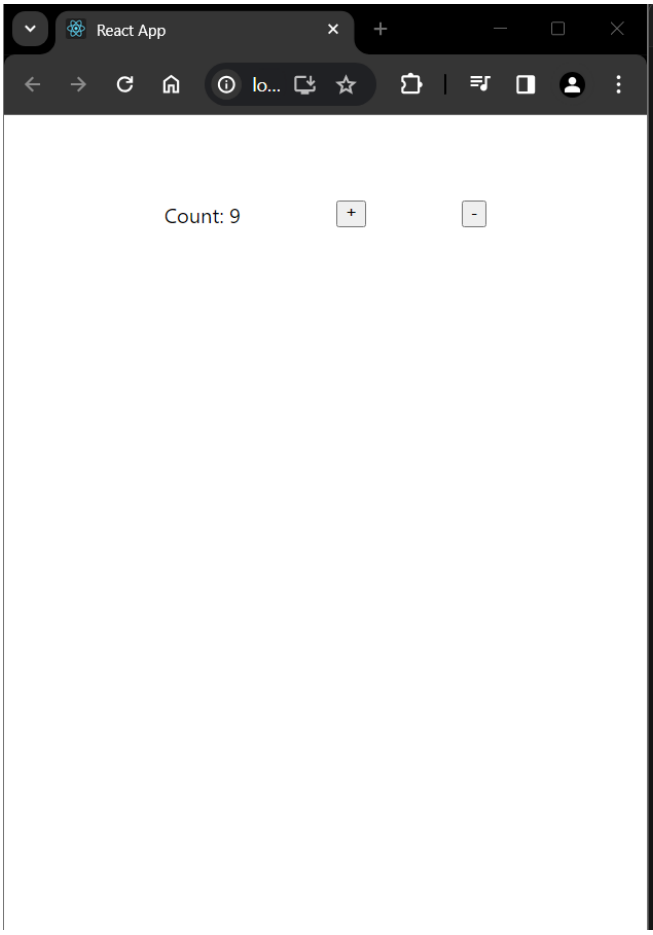
Code Flow:

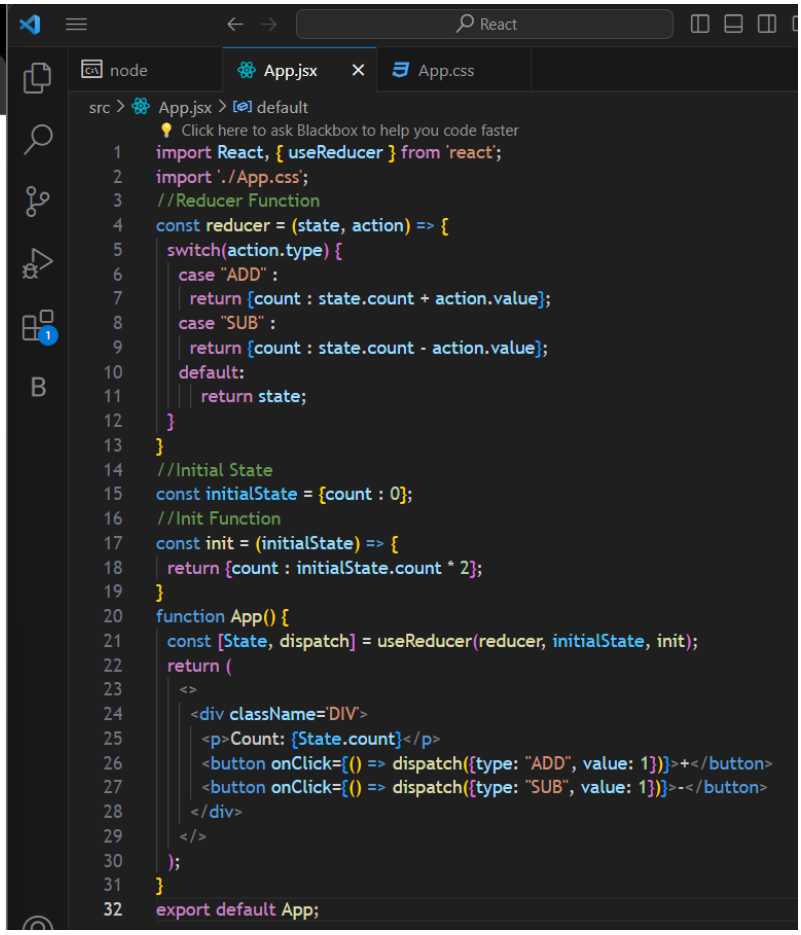
1. The initial state is `{ count: 0 }`. The Counter component renders, displaying the count value and two buttons for incrementing and decrementing.

2. When the "Increment" button is clicked, it calls `dispatch({ type: 'INCREMENT' })`. useReducer calls the reducer function with the current state `{ count: 0 }` and the action `{ type: 'INCREMENT' }`.

3. The reducer function increments the count value and returns `{ count: 1 }`. useReducer updates the state to `{ count: 1 }` and re-renders the component.

4. The updated count value is displayed, and the process repeats for decrementing and subsequent increments.





1. useReducer is called with reducer, initialState, and init, and returns state and dispatch. The init function is used to initialize the state. In this case, it doubles the initial count value.

2. The reducer function handles ADD and SUBTRACT actions, updating the count value based on the action type and value.

3. The Counter component renders a count value from state and two buttons that dispatch ADD and SUBTRACT actions with a value of 1.

4. When a button is clicked, dispatch is called with the corresponding action type and value, which updates the state using the reducer. The component re-renders with the updated count value.

<u>useHistory Hook:</u>

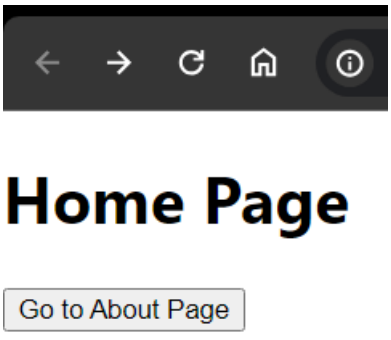
The useHistory hook is a feature provided by the react-router-dom library in React. It allows you to access the history object, which contains information about the current session's navigation history. This history object provides methods to navigate programmatically, such as push to add a new entry to the history stack, replace to replace the current entry, go to navigate to a specific entry in the history stack, and goBack and goForward to move backward or forward through the history stack.

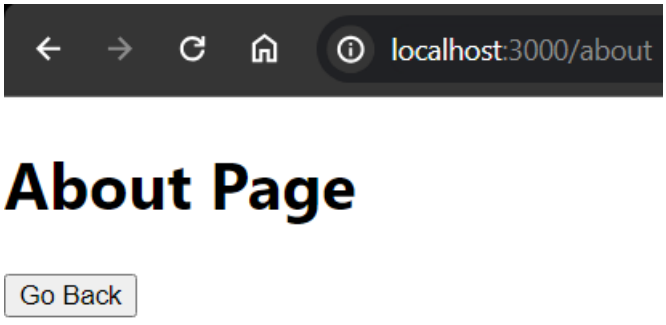
By using the useHistory hook, you can access the history object in functional components and perform navigation based on user interactions or other conditions within your application. This enables you to create dynamic and interactive user interfaces in React applications that utilize routing.

```jsx
import React from 'react';
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';
import Home from './Home';
import About from './About';
const App = () => {
  return (
    <Router>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
        <Route path="*" element={<h1>Not Found</h1>} />
      </Routes>
    </Router>
  );
};
export default App;
```

```jsx
import React from 'react';
import {useNavigate } from 'react-router-dom';
const Home = () => {
  const navigate = useNavigate();
  const handleClick = () => {
    navigate('/about');
  };
  return (
    <div>
      <h1>Home Page</h1>
      <button onClick={handleClick}>Go to About Page</button>
    </div>
  );
};
export default Home;
```

```jsx
import React from 'react';
import { useNavigate } from 'react-router-dom';
const About = () => {
  const navigate = useNavigate();
  const handleClick = () => {
    navigate(-1);
  };
  return (
    <div>
      <h1>About Page</h1>
      <button onClick={handleClick}>Go Back</button>
    </div>
  );
};
export default About;
```

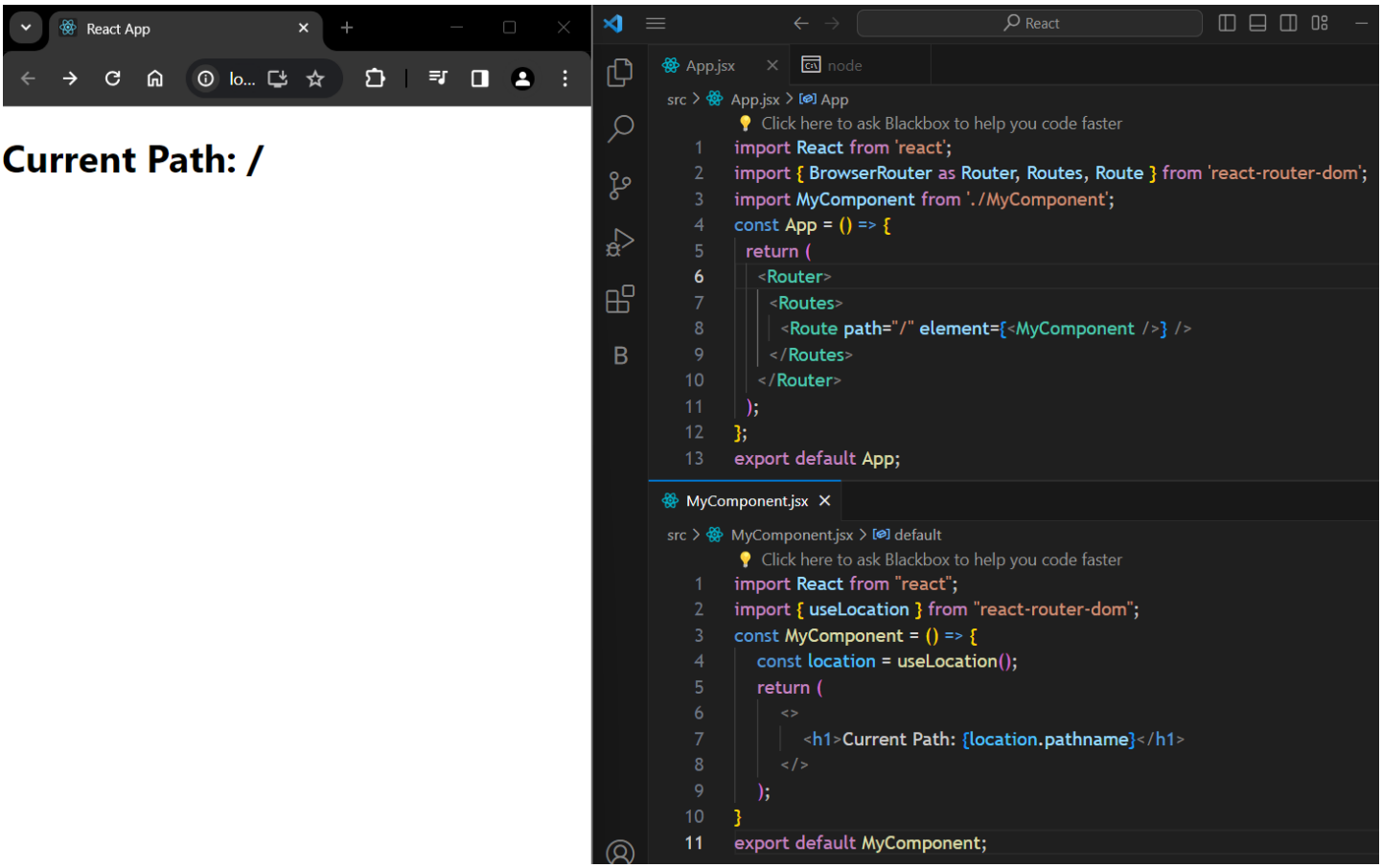




useLocation Hook:

The useLocation hook is a React Router hook that provides access to the current location object in your application. This location object contains information about the current URL, including the pathname, search parameters, hash, and state.

When you use the useLocation hook, React Router subscribes your component to the router's location changes. This means that whenever the URL changes (e.g., when a user navigates to a different route), your component will re-render with the updated location object.



You can now access various properties of the location object, such as location.pathname, location.search, location.hash, and location.state.

console.log(location.pathname); // Current pathname

console.log(location.search); // Query string parameters

console.log(location.hash); // URL hash

console.log(location.state); // State object (if any)

You can use the location data to conditionally render different content based on the current URL or perform other actions based on the URL.