# Frontend Development

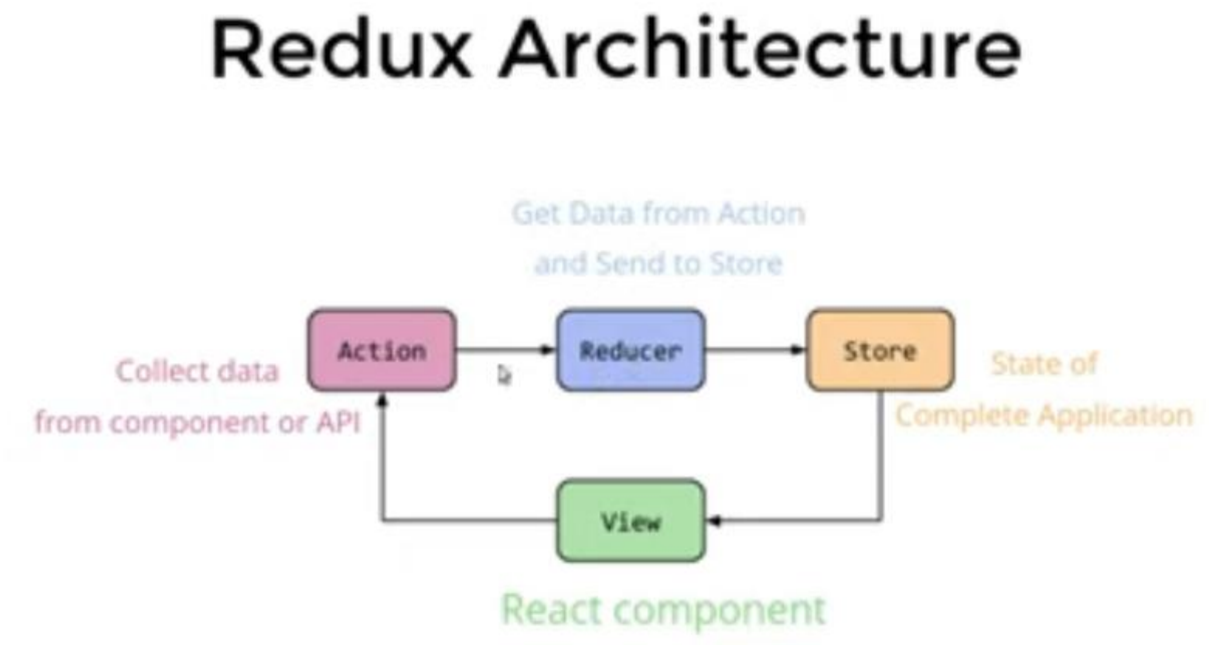
## React Redux:

A container where you can store your whole application data. So we call it to state management. It doesn't belong to the component state.

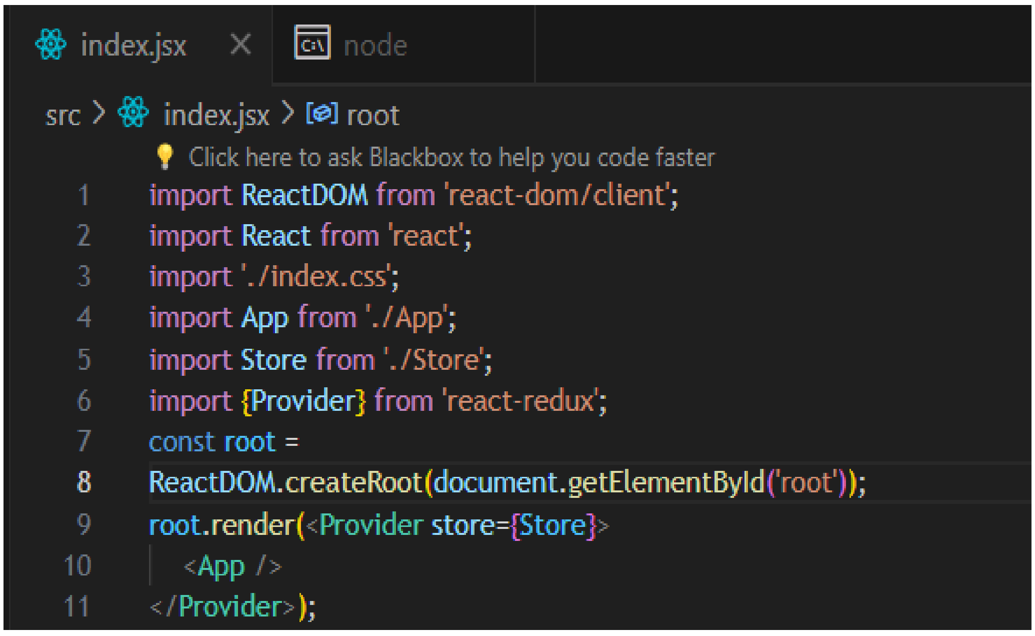
When a JavaScript application grows big. It becomes difficult for the user to manage its state. Redux solves this problem by managing application's state with a single global object called Store. React Redux makes testing very easy. Consistency throughout the application.

## Actions & Reducers:

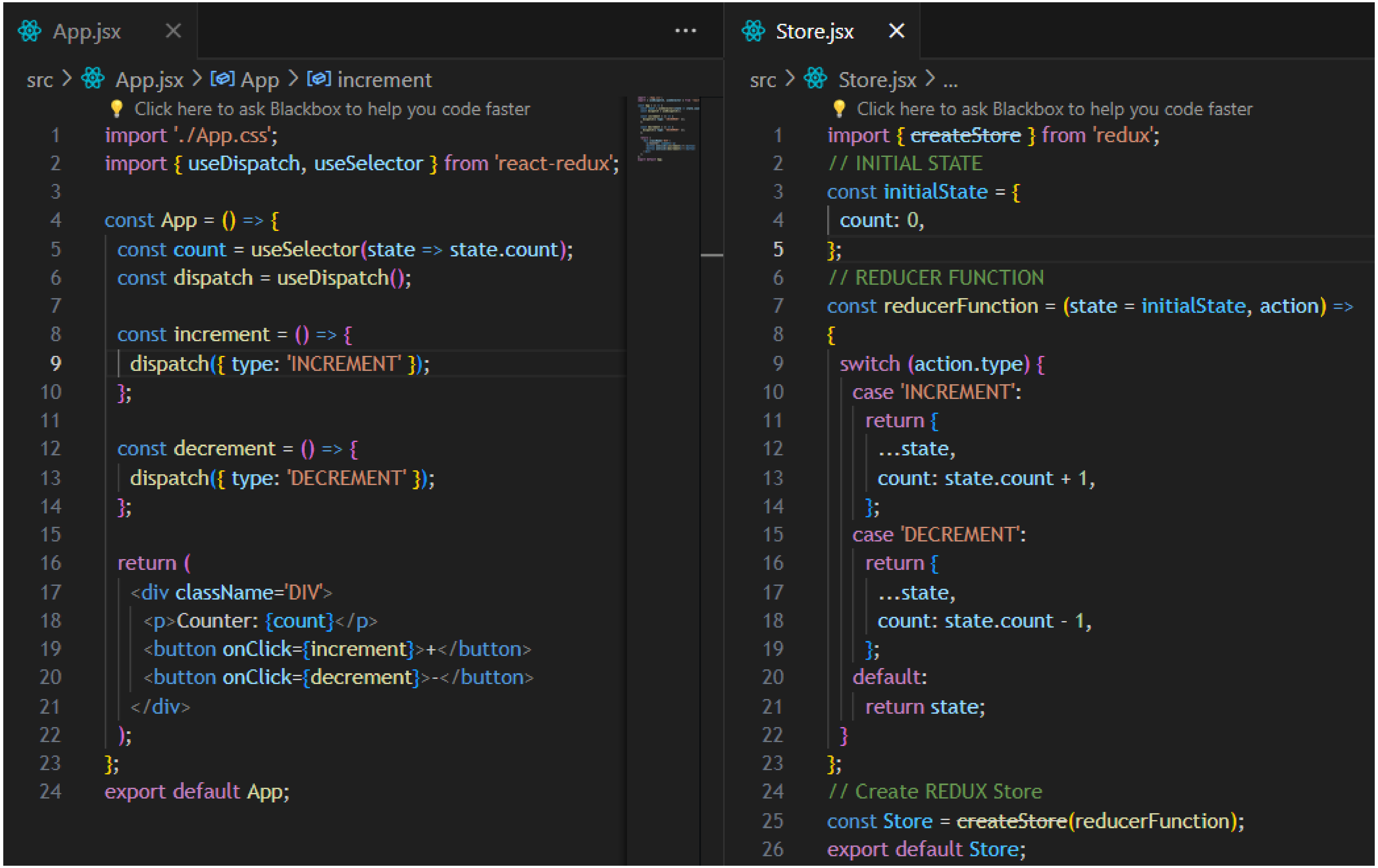
An Action is a plain object that describes the intention to cause change. A Reducer is a function that determines changes to an application's state. Return the new state and tell the store how to do. It uses the action it receives to determine this change.


Redux Architecture
Get Data from Action and Send to Store
Collect data from component or API
Action
Reducer
Store
State of Complete Application
View
React component

npm install redux npm install react-redux



```jsx
import ReactDOM from 'react-dom/client';
import React from 'react';
import './index.css';
import App from './App';
import Store from './Store';
import {Provider} from 'react-redux';
const root =
ReactDOM.createRoot(document.getElementById('root'));
root.render(<Provider store={Store}>
    <App />
</Provider>);
```



App.jsx
src > App.jsx > App > increment
Click here to ask Blackbox to help you code faster

```jsx
import './App.css';
import { useDispatch, useSelector } from 'react-redux';

const App = () => {
  const count = useSelector(state => state.count);
  const dispatch = useDispatch();

  const increment = () => {
    dispatch({ type: 'INCREMENT' });
  };

  const decrement = () => {
    dispatch({ type: 'DECREMENT' });
  };

  return (
    <div className='DIV'>
      <p>Counter: {count}</p>
      <button onClick={increment}>+</button>
      <button onClick={decrement}>-</button>
    </div>
  );
};
export default App;
```

Store.jsx
src > Store.jsx > ...
Click here to ask Blackbox to help you code faster

```jsx
import { ~~createStore~~ } from 'redux';
// INITIAL STATE
const initialState = {
  count: 0,
};
// REDUCER FUNCTION
const reducerFunction = (state = initialState, action) =>
{
  switch (action.type) {
    case 'INCREMENT':
      return {
        ...state,
        count: state.count + 1,
      };
    case 'DECREMENT':
      return {
        ...state,
        count: state.count - 1,
      };
    default:
      return state;
  }
};
// Create REDUX Store
const Store = ~~createStore~~(reducerFunction);
export default Store;
```

Here's a detailed explanation of the provided code, broken down into three files: `index.js`, `App.js`, and `Store.js`. I'll use bullet points for clarity:

**index.js**:

- Import Statements:

  - Imports `ReactDOM` from `react-dom/client`, `React` from `react`, and other necessary components and stylesheets.

- Root Element:

  - Creates a root element using `ReactDOM.createRoot` and specifies the element with the id of `'root'` in the `document`.

- Rendering:

  - Uses the `root.render` method to render the `App` component wrapped in a `Provider` component from `react-redux`. The `Provider` component provides the Redux store to the entire application.

**App.js:**

- Import Statements:

  - Imports necessary components and stylesheets.

- Functional Component `App`:

  - Defines a functional component named `App` that represents the main application.

  - Uses `useSelector` hook to access the `count` state from the Redux store.

  - Uses `useDispatch` hook to dispatch actions to the Redux store.

- Render Method:

  - Renders a `div` element containing the current count value and two buttons for incrementing and decrementing the count.

  - The `onClick` handlers for the buttons dispatch `INCREMENT` and `DECREMENT` actions, respectively.

**Store.js:**

- Import Statement:

  - Imports `createStore` function from `redux`.

- Initial State:

  - Defines an initial state object with a `count` property set to `0`.

- Reducer Function:

- Defines a reducer function that takes the current state and an action, and returns a new state based on the action type.

  - The reducer handles `INCREMENT` and `DECREMENT` actions by updating the `count` property in the state.

- Redux Store Creation:

  - Uses the `createStore` function to create a Redux store, passing the reducer function as an argument.

- Export Statement:

  - Exports the created Redux store.

**Data Flow:**

1. Initialization:

   - The `Store.js` file initializes the Redux store with an initial state.

2. Rendering:

   - The `index.js` file renders the `App` component wrapped in a `Provider` component, providing the Redux store to the application.

3. State Access:

   - The `App` component uses `useSelector` to access the `count` state from the Redux store.

4. Dispatching Actions:

   - The `App` component uses `useDispatch` to dispatch `INCREMENT` and `DECREMENT` actions to the Redux store when the buttons are clicked.

5. State Update:

   - When an action is dispatched, the reducer in `Store.js` updates the state based on the action type.

6. Re-rendering:

   - The component re-renders with the updated state, displaying the new count value.

**Redux Tool Kit:**

React Redux Toolkit is the official, recommended way to write Redux logic. It provides a set of tools and best practices to simplify Redux development, making it more efficient and scalable. To install the latest version of React Redux Toolkit in a React project, you can use npm or yarn:

npm install @reduxjs/toolkit react-redux

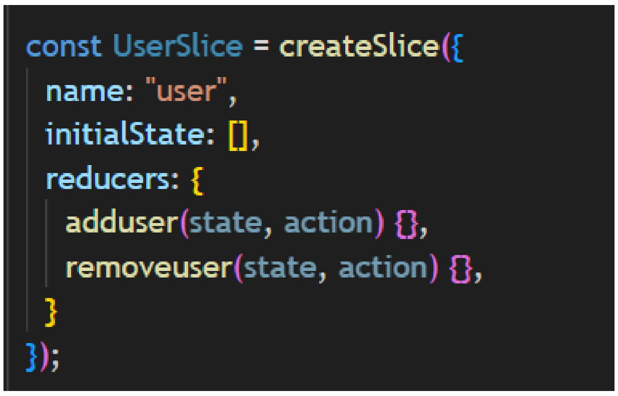
With specifically recommend that our users should use the redux toolkit and should not use the legacy redux core package for any new redux code today. We want all redux users to write their redux code with the redux toolkit because it simplifies your code and eliminates many common redux mistakes and bugs!

**Slices In Redux Tool Kit:**

A function that accepts a slice name, initial State, and Object of reducer functions, and automatically generates action creators and action types that correspond to the reducers and state.

createSlice is a function provided by Redux Toolkit that helps in reducing boilerplate code when creating Redux slices, which are pieces of state and logic in a Redux store. It combines the definition of a slice's initial state, reducer functions, and action creators into a single, concise syntax. In Redux, a slice is a collection of reducer logic and actions for a specific slice of your application's state.

createSlice allows you to define a slice more intuitively, by specifying the initial state and a set of reducer functions that define how the state can be updated in response to actions.

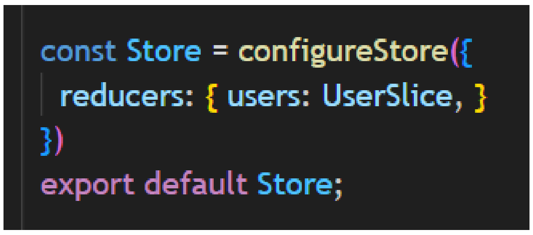


```
const UserSlice = createSlice({
  name: "user",
  initialState: [],
  reducers: {
    adduser(state, action) {},
    removeuser(state, action) {},
  }
});
```

**Store In React Redux Tool Kit:**

configureStore is a function provided by Redux Toolkit that simplifies the setup of a Redux store. It combines several Redux-related functions into a single function call, making it easier to create a store with commonly used configurations. In Redux, a store holds the whole state tree of your application.

The only way to change the state inside it is to dispatch an action. The configureStore function provides a way to create a Redux store with pre-configured settings, including middleware, reducers, and dev tools setup.



```
const Store = configureStore({
  reducers: { users: UserSlice, }
})
export default Store;
```

**Provider:**

The Provider component from react-redux is used to provide the Redux store to the entire application. It wraps the BrowserRouter and App components to ensure that the Redux store is available to all components.

**useSelector:**

useSelector is a hook provided by React Redux that allows components to extract data from the Redux store state. In the App component, isAuth is used to determine if the user is authenticated and conditionally render the Expense component based on the authentication status.

useSelector is a hook provided by React-Redux that allows functional components to extract and read data from the Redux store's state. It provides a way to access the Redux store's state without having to subscribe to the store manually or pass the state down through the component hierarchy.

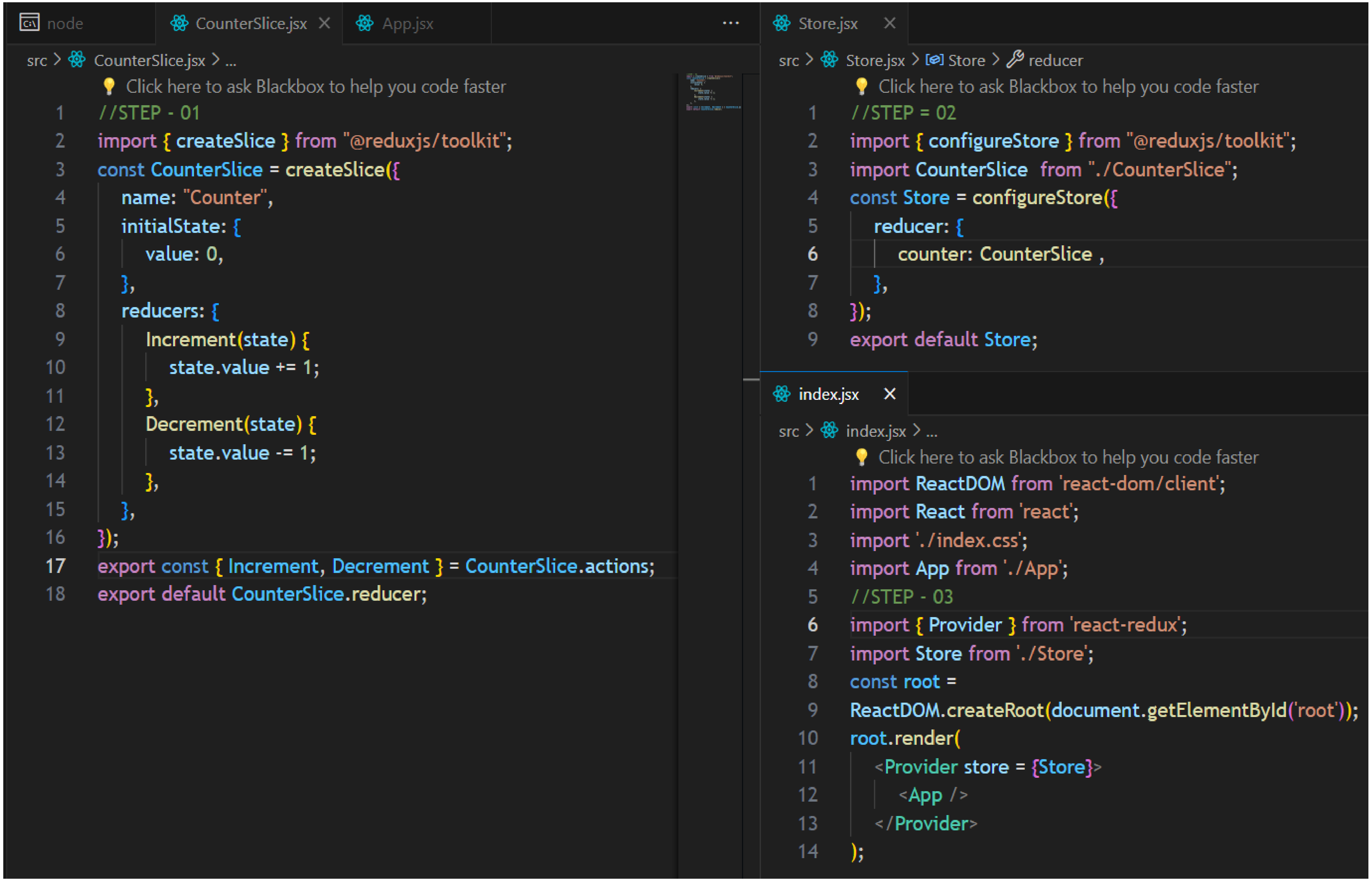
In Redux, the getState function is used to access the current state of the Redux store. However, when using React-Redux, you can use the useSelector hook to select and extract specific pieces of state from the Redux store's state tree.

**useDispatch:**

useDispatch is another hook provided by React Redux that returns the Redux store's dispatch function. This hook is used to dispatch actions to the Redux store. In the Header component, dispatch is used to dispatch the logout action when the user clicks the "Logout" button.

useDispatch is a hook provided by React-Redux that allows functional components to dispatch actions to the Redux store. It provides a way to interact with the Redux store without having to explicitly pass the dispatch function down through the component hierarchy.

In Redux, the dispatch function is used to send actions to the Redux store. It is typically accessed through the store. dispatch method. However, when using React-Redux, you can use the useDispatch hook to access the dispatch function directly within your functional components.



node | CounterSlice.jsx | App.jsx

src > CounterSlice.jsx > ...

💡 Click here to ask Blackbox to help you code faster

```jsx
//STEP - 01
import { createSlice } from "@reduxjs/toolkit";
const CounterSlice = createSlice({
  name: "Counter",
  initialState: {
    value: 0,
  },
  reducers: {
    Increment(state) {
      state.value += 1;
    },
    Decrement(state) {
      state.value -= 1;
    },
  },
});
export const { Increment, Decrement } = CounterSlice.actions;
export default CounterSlice.reducer;
```

Store.jsx

src > Store.jsx > Store > reducer

💡 Click here to ask Blackbox to help you code faster

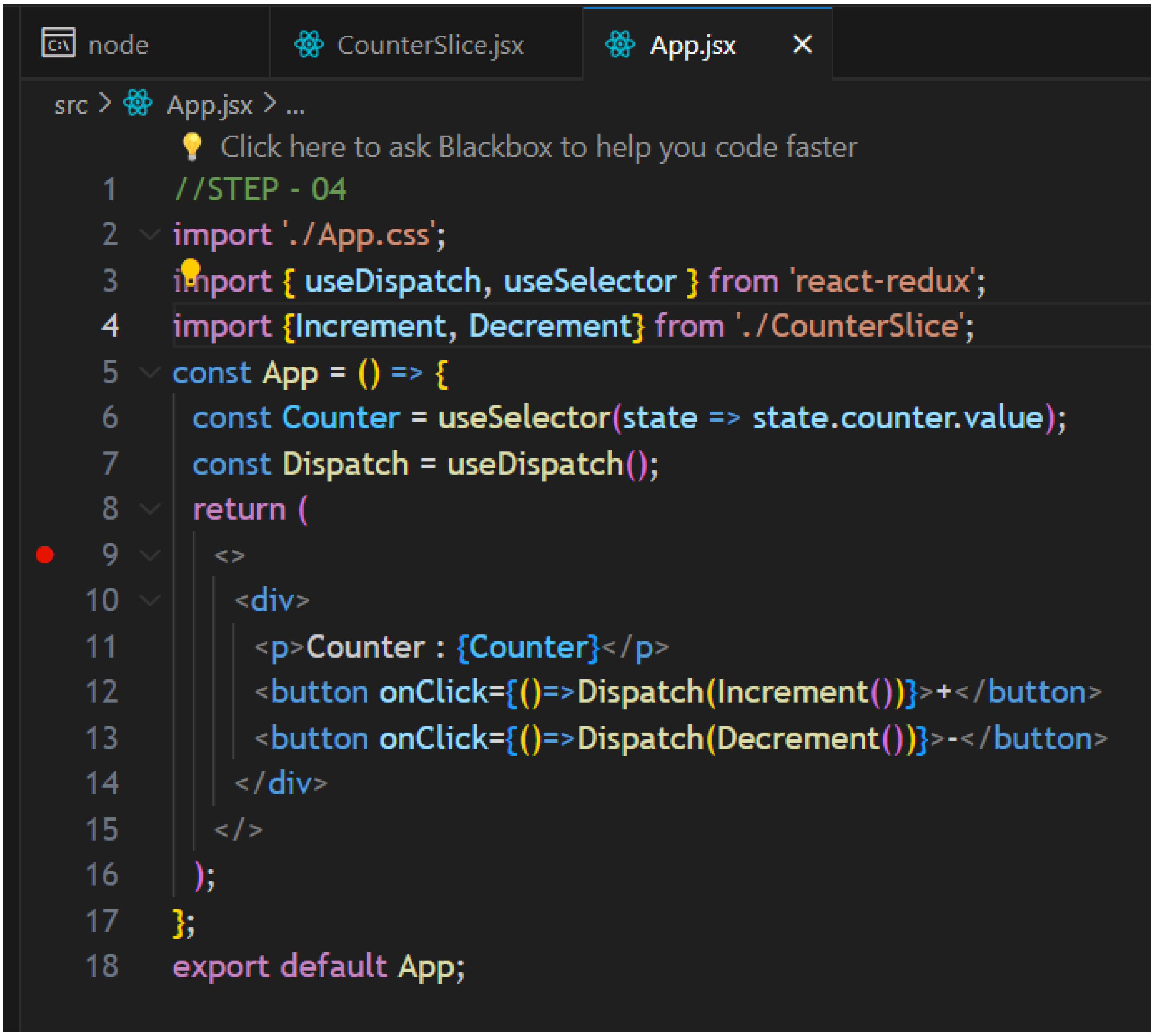
```jsx
//STEP = 02
import { configureStore } from "@reduxjs/toolkit";
import CounterSlice  from "./CounterSlice";
const Store = configureStore({
  reducer: {
    counter: CounterSlice ,
  },
});
export default Store;
```

index.jsx

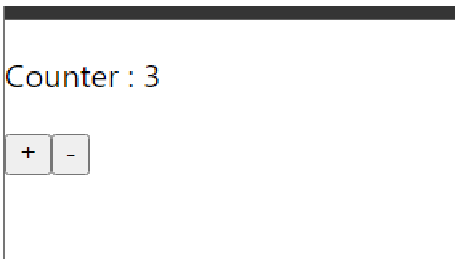
src > index.jsx > ...

💡 Click here to ask Blackbox to help you code faster

```jsx
import ReactDOM from 'react-dom/client';
import React from 'react';
import './index.css';
import App from './App';
//STEP - 03
import { Provider } from 'react-redux';
import Store from './Store';
const root =
ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <Provider store = {Store}>
    <App />
  </Provider>
);
```



```jsx
//STEP - 04
import './App.css';
import { useDispatch, useSelector } from 'react-redux';
import {Increment, Decrement} from './CounterSlice';
const App = () => {
  const Counter = useSelector(state => state.counter.value);
  const Dispatch = useDispatch();
  return (
    <>
      <div>
        <p>Counter : {Counter}</p>
        <button onClick={()=>Dispatch(Increment())}>+</button>
        <button onClick={()=>Dispatch(Decrement())}>-</button>
      </div>
    </>
  );
};
export default App;
```



Counter : 3

**Component Interaction:** In App.js, the Counter component renders a counter value and two buttons for incrementing and decrementing the counter.

**Redux Store Setup:** The Redux store is configured in store.js using configureStore from @reduxjs/toolkit. It includes the counterReducer from counterSlice.js.

**State Access and Dispatch:** The Counter component uses useSelector to access the counter state value from the Redux store. It also uses useDispatch to get a reference to the store's dispatch function.

**Action Dispatch:** When the user clicks the "Increment" or "Decrement" button, the corresponding action (increment or decrement) is dispatched to the Redux store using the dispatch function obtained from useDispatch.

**Reducer Execution:** In counterSlice.js, the counterSlice defines the initial state of the counter and two reducers (increment and decrement) to update the counter state. When an action is dispatched, the corresponding reducer is executed, updating the counter value in the store.

**State Update:** As the reducer modifies the state, the Redux store notifies all subscribed components, including App.js, of the state change. The Counter component automatically re-renders with the updated counter value fetched using useSelector.