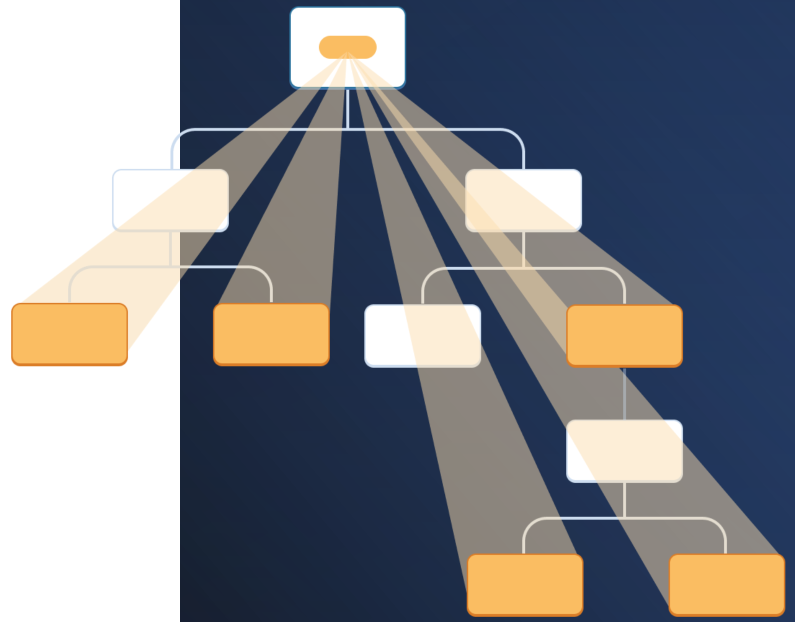
ReactJS Context API and Redux

Thusithanjana Thilakarathna

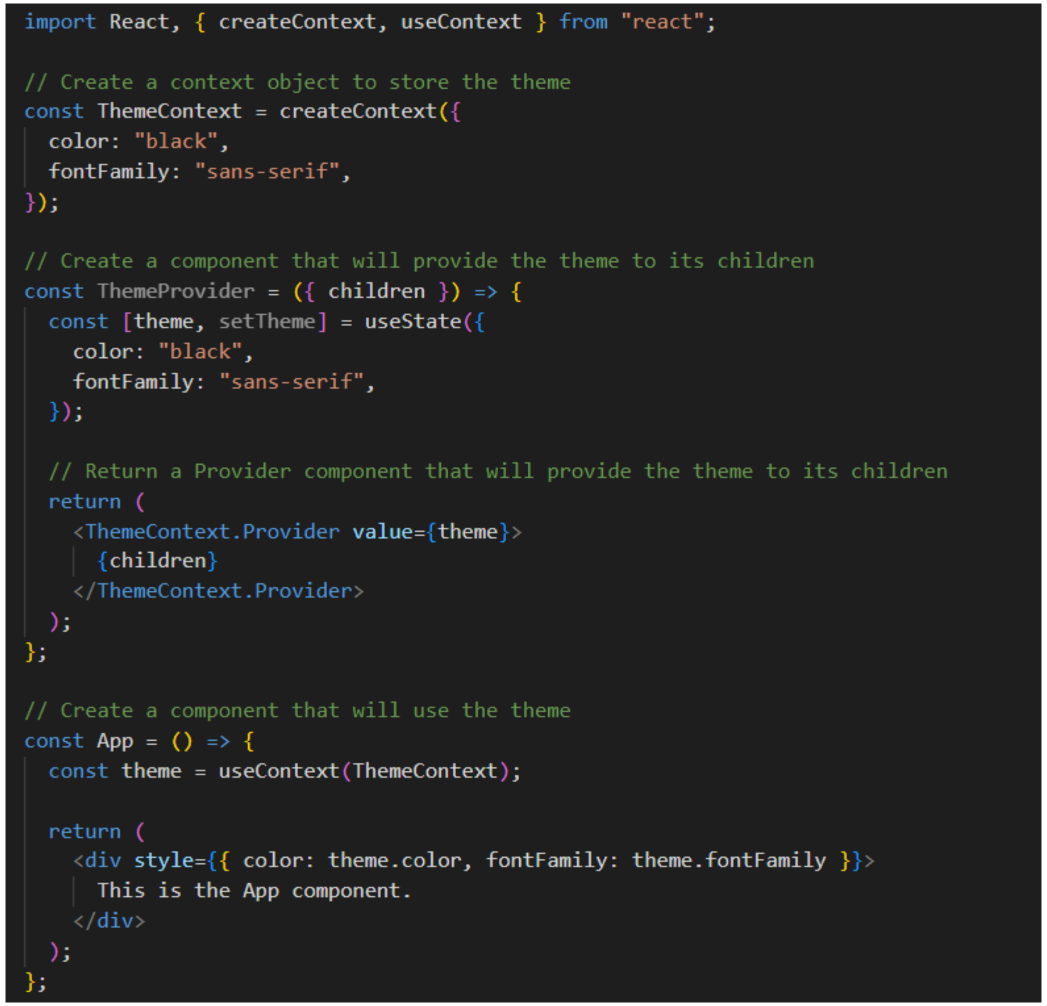
Introduction to Context API

- Data will be passed parent to child components through props
- Passing props through many components will be verbose and inconvenient
- Context lets the parent component make some information available to any component in the tree below it—no matter how deep—without passing it explicitly through props



How to use the Context API?

- To use the Context API, you first need to create a context object. This object will contain the data that you want to share. You can then use the useContext hook to access the data in the context object from your components.



```
import React, { createContext, useContext } from "react";

// Create a context object to store the theme
const ThemeContext = createContext({
  color: "black",
  fontFamily: "sans-serif",
});

// Create a component that will provide the theme to its children
const ThemeProvider = ({ children }) => {
  const [theme, setTheme] = useState({
    color: "black",
    fontFamily: "sans-serif",
  });

  // Return a Provider component that will provide the theme to its children
  return (
    <ThemeContext.Provider value={theme}>
      {children}
    </ThemeContext.Provider>
  );
};

// Create a component that will use the theme
const App = () => {
  const theme = useContext(ThemeContext);

  return (
    <div style={{ color: theme.color, fontFamily: theme.fontFamily }}>
      This is the App component.
    </div>
  );
};
```

Benefits of using the Context API

Reduced prop drilling: The Context API can help to reduce prop drilling, which is the practice of passing props down through the component tree. This can make your code more organized and easier to read.

Improved performance: The Context API can improve the performance of your React application by reducing the number of re-renders. This is because the Context API only updates the components that need to be updated, instead of the entire component tree.

Easier to test: The Context API can make it easier to test your React application. This is because you can test the data in the context object directly, instead of testing each component that uses the context object.

Examples of using the Context API

Sharing a theme: You can use the Context API to share a theme between all of the components in your application. This will make it easy to change the look and feel of your application without having to update each component individually.

Managing user state: You can use the Context API to manage user state, such as the current user's name and email address. This will make it easy to keep track of user information and update it as needed.

Storing application data: You can use the Context API to store application data, such as the current time and date. This will make it easy to access application data from anywhere in your application.

Introduction to Redux

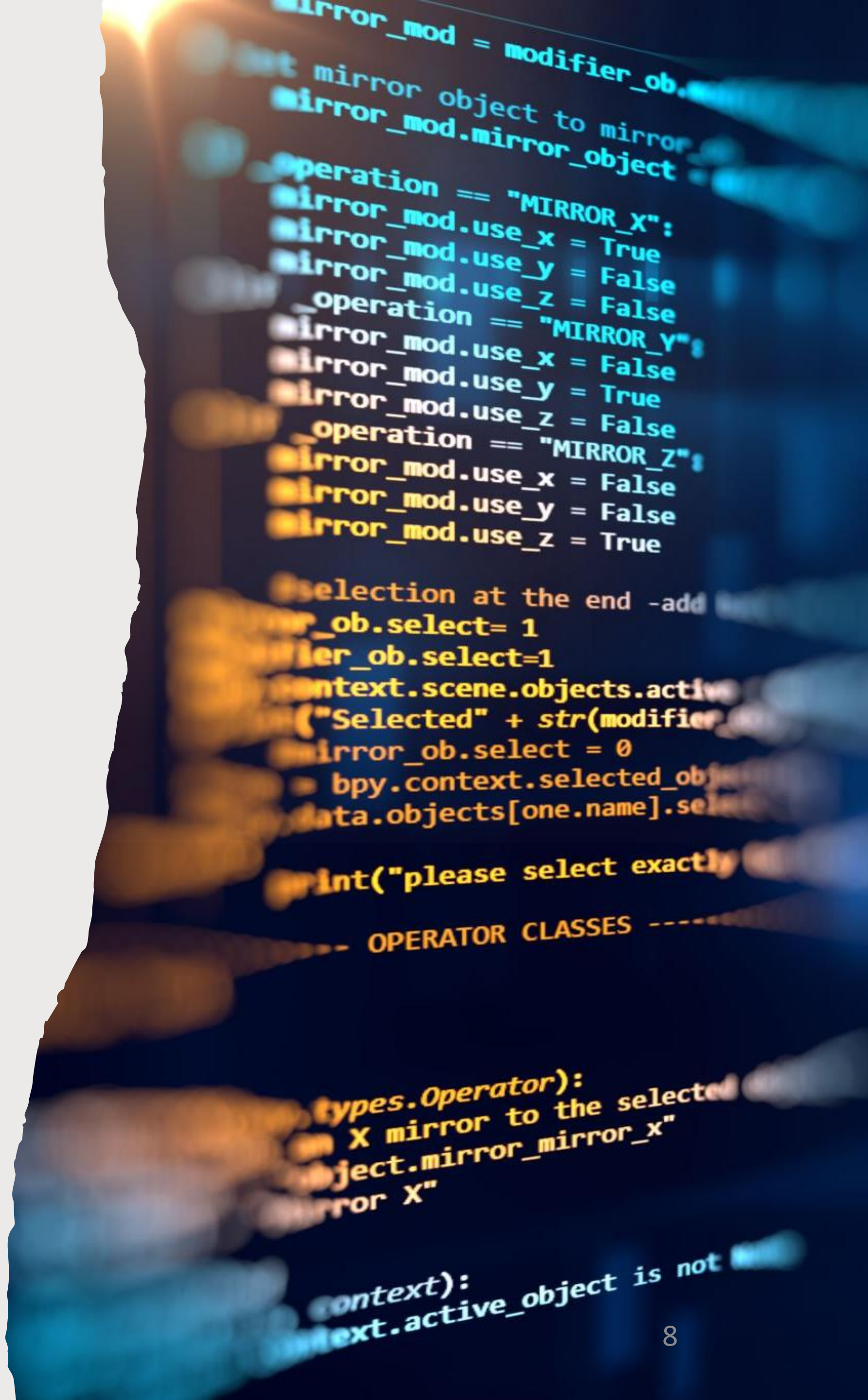
- Redux is a predictable state container for JavaScript apps. It helps you write applications that behave consistently, run in different environments (client, server, and native), and are easy to test.
- Centralized state management: Redux provides a single source of truth for your application state. This makes it easy to track changes to your state and to debug your application.
- Predictable state changes: Redux uses a single reducer to handle all state changes. This ensures that your state changes are predictable and easy to understand.
- Easy to test: Redux is easy to test because it provides a clear separation between your application state and your application logic.

Why Use React Redux?

- Redux itself is a standalone library that can be used with any UI layer or framework, including React, Angular, Vue, Ember, and vanilla JS. Although Redux and React are commonly used together, they are independent of each other.
- If you are using Redux with any kind of UI framework, you will normally use a "UI binding" library to tie Redux together with your UI framework, rather than directly interacting with the store from your UI code.
- React Redux is the official Redux UI binding library for React. If you are using Redux and React together, you should also use React Redux to bind these two libraries.

Integrating Redux with a UI

- Create a Redux store
- Subscribe to updates
- Inside the subscription callback:
    - Get the current store state
    - Extract the data needed by this piece of UI
    - Update the UI with the data
- If necessary, render the UI with initial state
- Respond to UI inputs by dispatching Redux actions

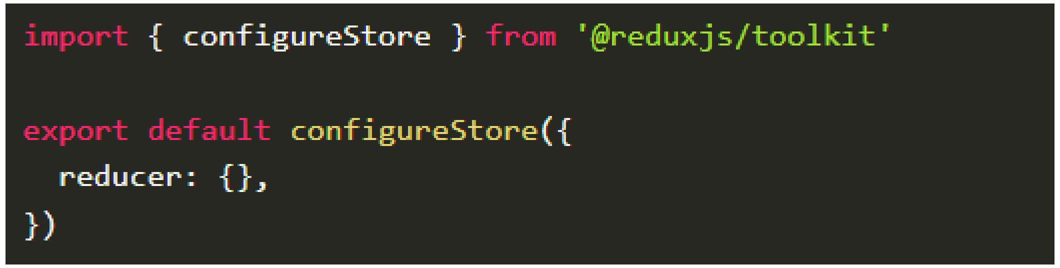


Setting up Redux

- npm install @reduxjs/toolkit react-redux

1. Create Redux Store

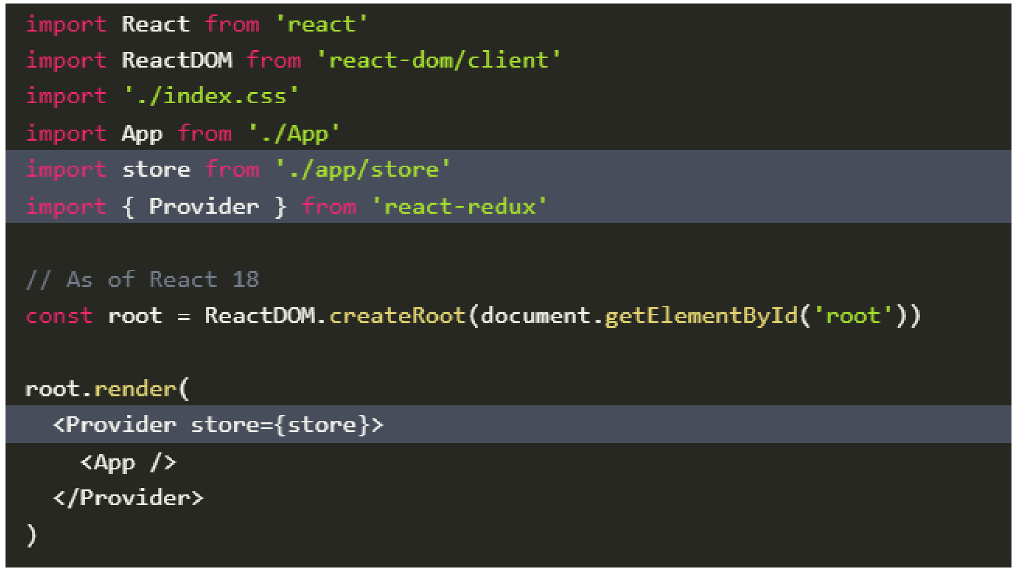
- Create a file named src/app/store.js. Import the configureStore API from Redux Toolkit. We'll start by creating an empty Redux store, and exporting it:



```
import { configureStore } from '@reduxjs/toolkit'

export default configureStore({
  reducer: {},
})
```

2. Provide the Redux Store to React

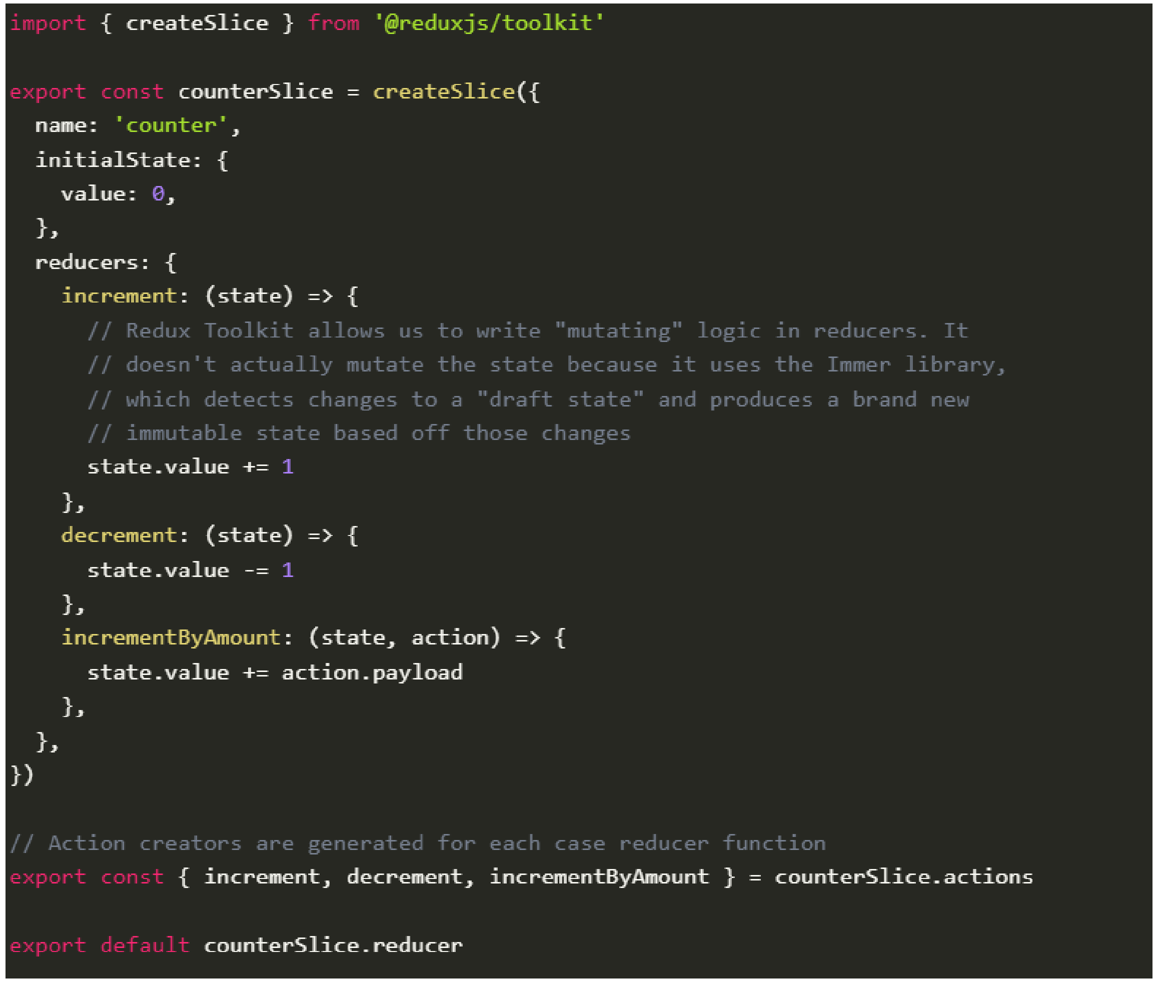
- Once the store is created, we can make it available to our React components by putting a React Redux <Provider> around our application in src/index.js. Import the Redux store we just created, put a <Provider> around your <App>, and pass the store as a prop:



```
import React from 'react'
import ReactDOM from 'react-dom/client'
import './index.css'
import App from './App'
import store from './app/store'
import { Provider } from 'react-redux'

// As of React 18
const root = ReactDOM.createRoot(document.getElementById('root'))

root.render(
  <Provider store={store}>
    <App />
  </Provider>
)
```

3. Create a Redux State Slice

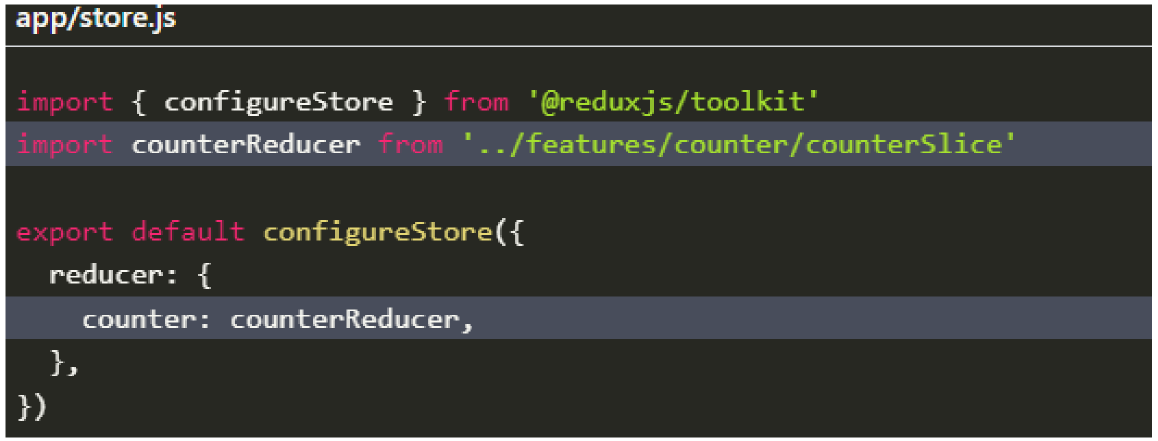
- Creating a slice requires a string name to identify the slice, an initial state value, and one or more reducer functions to define how the state can be updated.
- Once a slice is created, we can export the generated Redux action creators and the reducer function for the whole slice.
- Redux requires that we write all state updates immutably, by making copies of data and updating the copies.
- However, Redux Toolkit's createSlice and createReducer APIs use Immer inside to allow us to write "mutating" update logic that becomes correct immutable updates.



```js
import { createSlice } from '@reduxjs/toolkit'

export const counterSlice = createSlice({
  name: 'counter',
  initialState: {
    value: 0,
  },
  reducers: {
    increment: (state) => {
      // Redux Toolkit allows us to write "mutating" logic in reducers. It
      // doesn't actually mutate the state because it uses the Immer library,
      // which detects changes to a "draft state" and produces a brand new
      // immutable state based off those changes
      state.value += 1
    },
    decrement: (state) => {
      state.value -= 1
    },
    incrementByAmount: (state, action) => {
      state.value += action.payload
    },
  },
})

// Action creators are generated for each case reducer function
export const { increment, decrement, incrementByAmount } = counterSlice.actions

export default counterSlice.reducer
```

4. Add Slice Reducers to the Store

- Next, we need to import the reducer function from the counter slice and add it to our store.
- By defining a field inside the reducers parameter, we tell the store to use this slice reducer function to handle all updates to that state.

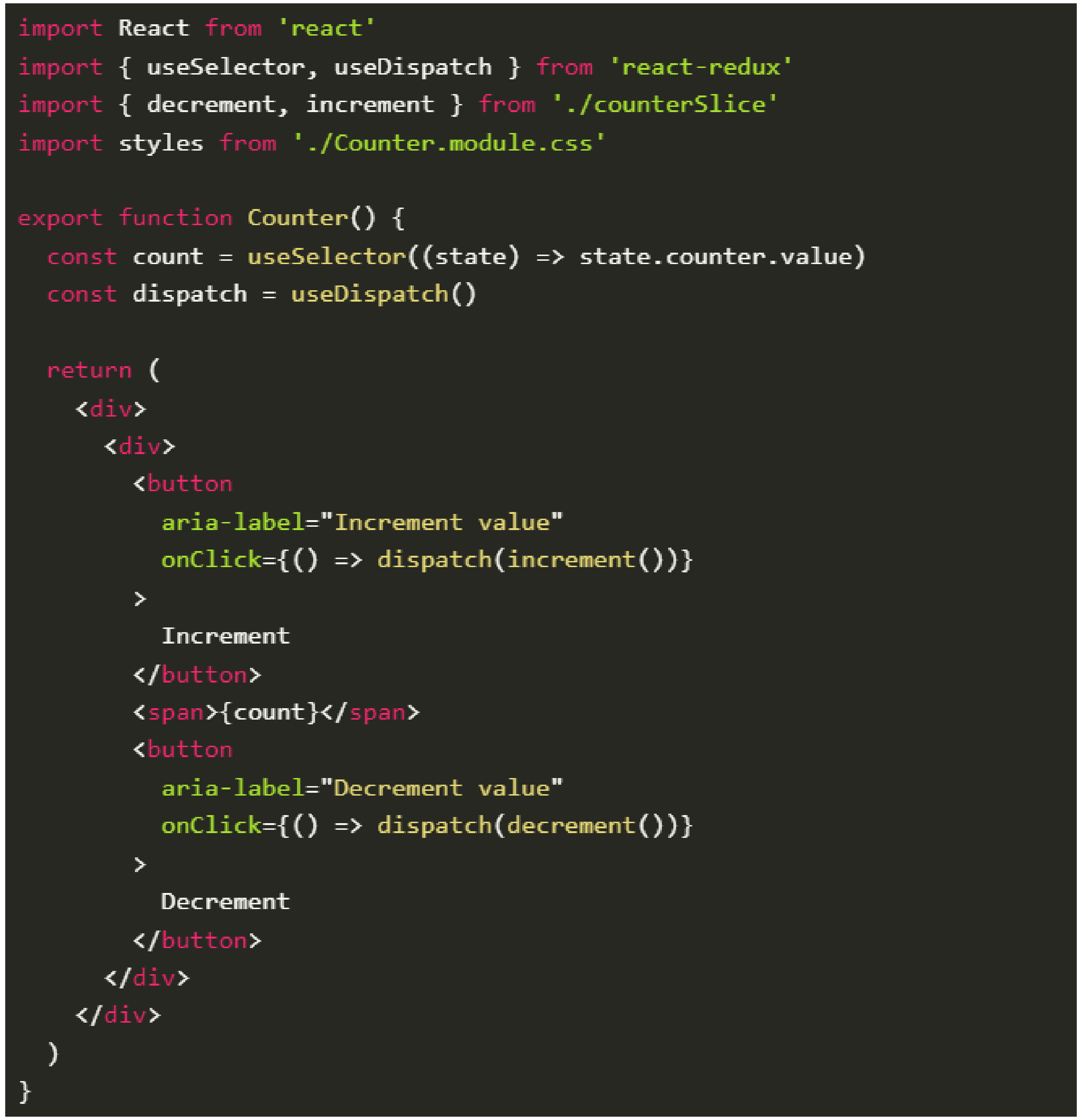


app/store.js

```
import { configureStore } from '@reduxjs/toolkit'
import counterReducer from '../features/counter/counterSlice'

export default configureStore({
  reducer: {
    counter: counterReducer,
  },
})
```

5. Use Redux State and Actions in React Components

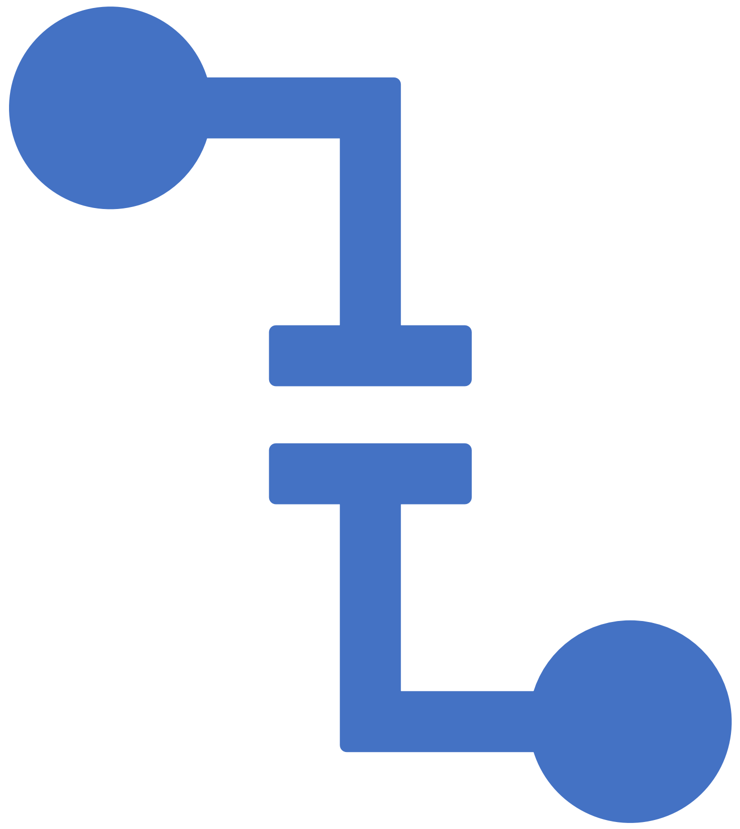
- Now we can use the React Redux hooks to let React components interact with the Redux store.
- We can read data from the store with useSelector, and dispatch actions using useDispatch.



```
import React from 'react'
import { useSelector, useDispatch } from 'react-redux'
import { decrement, increment } from './counterSlice'
import styles from './Counter.module.css'

export function Counter() {
  const count = useSelector((state) => state.counter.value)
  const dispatch = useDispatch()

  return (
    <div>
      <div>
        <button
          aria-label="Increment value"
          onClick={() => dispatch(increment())}
        >
          Increment
        </button>
        <span>{count}</span>
        <button
          aria-label="Decrement value"
          onClick={() => dispatch(decrement())}
        >
          Decrement
        </button>
      </div>
    </div>
  )
}
```

Advance React

- Concurrent Mode
- Suspense
- Server-side Rendering



Concurrent Mode

- Concurrent mode is a new feature in React 18 that allows React to render multiple components at the same time.
- This can improve the performance of React applications by allowing them to continue to be responsive even when they are doing long tasks, such as fetching data from an API.
- It is important to use it carefully. If you use it incorrectly, it can lead to unexpected behavior.
- Only use concurrent mode when it is necessary. If your application is not CPU-bound, then you do not need to use concurrent mode.
- Be careful about the order in which you render components. If you render components in the wrong order, it can lead to unexpected behavior.
- Use the useDeferredValue hook to defer the rendering of components until they are needed. This can help to improve the performance of your application.

Suspense

- Suspense is a new feature in React 18 that allows you to manage asynchronous operations in your application.
- It lets you display a fallback UI while your application is waiting for data to load.
- Suspense is enabled by using the Suspense component. This component takes a child component as its only argument.
- The child component is the component that will be rendered when the data is loaded.
- If the data is not loaded yet, the Suspense component will render a fallback UI.
- The fallback UI can be anything you want, such as a loading spinner or a message telling the user to wait.
- Suspense is a powerful feature that can improve the user experience of your application. It can help to make your application more responsive and less frustrating for users.

Server-side rendering (SSR)

- SSR is a technique that renders a web page on the server rather than in the browser.
- This means that the server generates the HTML for the page, and the browser simply displays it.
- Benefits
    - Improved SEO: Search engines can crawl and index pages that are rendered on the server, which can improve your website's search engine ranking.
    - Faster page load times: Pages that are rendered on the server are typically loaded faster than pages that are rendered in the browser, because the browser does not have to download and render the JavaScript code.
    - Better user experience: Pages that are rendered on the server are typically more responsive than pages that are rendered in the browser, because the browser does not have to wait for the JavaScript code to download and execute before it can start displaying the page.
- Challenges
    - Increased development time: SSR can add some additional development time to your project, because you need to write server-side code to render the pages.
    - Increased server costs: SSR can increase your server costs, because you need to have a server that can handle the load of rendering pages.

SSR vs CSR

SSR

- Improved SEO
- Faster initial page load times
- Better perceived performance
- increased development time
- Increased server costs
- More complex to debug

CSR

- More interactive and dynamic pages
- More control over the user experience
- Can be slower for initial page load
- Can be more difficult to optimize for SEO

Examples

SSR

- A news website that needs to rank well in search engines
- A landing page that needs to load quickly
- A static website that doesn't need a lot of interactivity

CSR

- A social media app that needs to be responsive and interactive
- A game that needs to be smooth and responsive
- A web application that needs to be customized by the user

Thank you!