**Exercise: React Redux**

Redux is a predictable state container for JavaScript applications, primarily used with React. Its main goal is to manage the state of an application in a predictable and centralized manner, making it easier to develop and maintain complex applications. In some cases, new React Hooks such as useState can replace it.

The core idea behind Redux can be summarized in three fundamental principles:

**Single Source of Truth**: In Redux, the entire state of an application is stored in a single JavaScript object called the "store." This means that all components in the application share the same state. Having a single source of truth makes it easier to understand and debug the application, as there is no ambiguity about where the state resides.

**State is Read-Only and Immutable**: In Redux, the state is read-only, meaning components cannot directly modify it. The only way to change the state is by dispatching actions. Actions are plain JavaScript objects that describe the type of change to be made to the state. This immutability ensures that the state transitions are traceable and helps maintain a history of state changes, making it easier to debug and understand how the state evolves over time.

**Changes are Made by Pure Functions**: Redux uses pure functions called "reducers" to specify how the application's state should change in response to dispatched actions. Reducers take the current state and an action as input and return a new state as output, without modifying the existing state. This ensures that the state transitions are predictable and free from side effects, making the behavior of the application easier to understand and test.

By following these principles, Redux provides several benefits:

**Centralized State Management**: Having a single global state accessible by all components makes it easier to reason about how data flows through the application. It eliminates the need for complex data passing between components through props, making the codebase more maintainable.

**Easy Collaboration and Scalability**: With Redux, managing application state becomes easier for larger teams as everyone shares the same data structure and follows the same pattern for state management. Additionally, as the application grows in complexity, Redux provides a scalable solution for managing state and handling asynchronous actions through middleware.

**Testability**: Pure reducers and unidirectional data flow make it easier to write unit tests for Redux applications. By providing a known input and verifying the output, testing becomes more straightforward and less prone to unexpected side effects.

Overall, Redux simplifies state management in JavaScript applications by providing a clear set of principles and patterns. It promotes a predictable and centralized approach, making it easier to develop, debug, and maintain complex applications over time.

1. First create a project and install redux and react-redux in it.

npm install redux

npm install react-redux

1. Let’s create Counter.js
   1. Create a React component named `Counter` that connects to the Redux store using the `connect` function from the 'react-redux' library.
   2. Importing the necessary dependencies:

import React from 'react';

import { connect } from 'react-redux';

* 1. The `React` import is required to define a React component. The `connect` import is used to connect the component to the Redux store.
  2. Define the `Counter` component:

const Counter = ({ count, increment, decrement }) => {

return (

<div>

<h1>Counter: {count}</h1>

<button onClick={increment}>Increment</button>

<button onClick={decrement}>Decrement</button>

</div>

);

};

* 1. The `Counter` component is a functional component that displays the current count value from the Redux store and provides buttons to increment and decrement the count. The count value is received as a prop, along with the `increment` and `decrement` functions.
  2. Map state to component props:

const mapStateToProps = (state) => {

return {

count: state.count,

};

};

* 1. The `mapStateToProps` function is used to map the state from the Redux store to the props of the `Counter` component. In this case, it maps the `count` property from the store's state to the `count` prop of the component.
  2. Map dispatch actions to component props:

const mapDispatchToProps = (dispatch) => {

return {

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

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

};

};

* 1. The `mapDispatchToProps` function is used to map dispatch actions to the props of the `Counter` component. It creates two functions, `increment` and `decrement`, which dispatch the 'INCREMENT' and 'DECREMENT' actions, respectively, when called.
  2. Connecting the component to the Redux store:

export default connect(mapStateToProps, mapDispatchToProps)(Counter);

* 1. The `connect` function is used to connect the `Counter` component to the Redux store. It takes two parameters: `mapStateToProps` and `mapDispatchToProps`. The connected component is then exported as the default export.
  2. By connecting the `Counter` component to the Redux store, it can access the count value from the store's state and dispatch actions to modify the state. The component will automatically update whenever the state changes, reflecting the latest count value.

1. Let’s create the Store, store.js. This is a typical implementation of a store in a React Redux application. First import the necessary dependencies:

import { createStore } from 'redux';

* 1. This line imports the `createStore` function from the 'redux' library. It is used to create the Redux store.
  2. Define the reducer function:

const reducer = (state = { count: 0 }, action) => {

switch (action.type) {

case 'INCREMENT':

return { count: state.count + 1 };

case 'DECREMENT':

return { count: state.count - 1 };

default:

return state;

}

};

* 1. Here, a reducer function is defined. A reducer is a pure function that takes the current state and an action as parameters and returns a new state based on the action. In this case, the reducer handles two actions: 'INCREMENT' and 'DECREMENT'. When the 'INCREMENT' action is dispatched, it increments the count value in the state by 1, and when the 'DECREMENT' action is dispatched, it decrements the count value by 1. If none of these actions are dispatched, it returns the current state as is.
  2. Create the Redux store:

const store = createStore(reducer);

* 1. The `createStore` function is invoked with the reducer function as an argument to create the Redux store. The store holds the state tree of your application and provides methods to interact with it.
  2. Export the store:

export default store;

* 1. The created Redux store is exported as the default export of this file, allowing other parts of the application to import and use it.
  2. To use this store in a React component, import it and provide it to the application using a `<Provider>` component from the 'react-redux' library. Other components in the application can then access the store's state and dispatch actions using the `useSelector` and `useDispatch` hooks, respectively.

1. Let’s edit App.js. It serves as the entry point of your React application.
   1. Import the necessary dependencies:

import React from 'react';

import { Provider } from 'react-redux';

import Counter from './Counter';

import store from './store';

* 1. The `React` import is required to define a React component. The `Provider` import is used to wrap the application and provide access to the Redux store. The `Counter` import is the component we previously discussed. The `store` import is the Redux store we created.
  2. Define the `App` component:

const App = () => {

return (

<Provider store={store}>

<Counter />

</Provider>

);

};

* 1. The `App` component is a functional component that serves as the main container for your application. It wraps the `Counter` component with the `Provider` component from 'react-redux'. The `Provider` component allows all components in the component tree to access the Redux store.
  2. The `store` is passed as a prop to the `Provider` component, making it available to all components in the application.
  3. Export the `App` component:

export default App;

* 1. The `App` component is exported as the default export of this file, allowing it to be used in other parts of the application.
  2. By wrapping the `Counter` component with the `Provider` component and providing the Redux store, the `Counter` component and any other components within the `Provider` component's tree can access the Redux store's state and dispatch actions to modify the state. The `App` component acts as the top-level container for your Redux-powered React application.