React: how to use Redux

React is a JavaScript library that has revolutionized user interface development by making it easier to create reusable and responsive components. However, as applications grow in complexity, state management becomes an increasingly significant challenge. This is where Redux comes in as a powerful tool for managing state effectively and predictably.

What is Redux?

Redux is a state management library for JavaScript, often used with React, although it can be adopted with other frameworks or libraries. Its fundamental idea is to centralize the application state in a store, making it easier to manage and update the complex state.

Key Principles of Redux

1. **Store**: A store is an object that holds the state of the application. The only way to change state in a Redux app is to send an action, an object that describes what happened.

2. **Action**: Actions are flat objects that represent a change in state. They contain a type that identifies the type of action and, optionally, additional data.

3. **Reducer**: Reducers are pure functions that specify how the state of the application changes in response to an action. They receive the current state and an action, returning the new state.

4. **Dispatch**: The `dispatch` function is the way with which the shares are sent to the store. When you call `dispatch` with an action, the store calls your reducer with the provided action.

5. **Middleware**: Redux offers the possibility of using middleware to perform asynchronous actions, manage side effects or intervene during the dispatch phase.

How to integrate Redux into a React application

Installing the necessary packages

To use Redux in a React project, you need to install some packages. You can do this via npm or yarn:

```
npm install redux react-redux
# or
yarn add redux react-redux
```

Create a store

```
// store.js
import { createStore } from 'redux';
import rootReducer from './reducers'; // Import your
main reducer

const store = createStore(rootReducer);

export default store;
```

Define actions

```
// actions.js
export const increment = () => {
   return {
     type: 'INCREMENT',
   };
};

export const decrement = () => {
   return {
     type: 'DECREMENT',
   };
};
```

Create reducers

```
// reducers.js
const initialState = {
   count: 0,
};

const counterReducer = (state = initialState, action)
=> {
   switch (action.type) {
     case 'INCREMENT':
       return { count: state.count + 1 };
     case 'DECREMENT':
       return { count: state.count - 1 };
     default:
       return state;
   }
};

export default counterReducer;
```

Connecting React to Redux

```
// App.js
import React from 'react';
import { useSelector, useDispatch } from 'react-
redux';
import { increment, decrement } from './actions';

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

   return (
     <div>
       <h1>Counter: {count}</h1>
       <button onClick={() =>
dispatch(increment())}>Increment</button>
       <button onClick={() =>
dispatch(decrement())}>Decrement</button>
     </div>
   );
}

export default App;
```

Connect the store to the React application

```
// index.js
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import store from './store';
import App from './App';

ReactDOM.render(
   <Provider store={store}>
     <App />
   </Provider>,
   document.getElementById('root')
);
```

Conclusions

Integrating Redux into a React project may seem complicated at first, but once you understand the basics, it offers a robust and scalable way to manage application state. By centralizing state in a store and following the one-way flow of actions, Redux provides a clear and predictable architecture that facilitates the development and maintenance of complex applications.