31 Jan 2024 · Software Engineering

How To Integrate Redux with React Based Application: A Step By Step Tutorial

Written by:
Prasandeep

Reviewed by:
Dan Ackerson

13 min read

Contents

Redux is the most powerful state management tool in the React.Js library, and it's frequently used by Frontend Engineers to build complex applications. With the help of Redux, we can easily manage the state of our React applications, which helps us to develop and maintain complex applications.

In this tutorial, we will discuss how to integrate Redux with React applications. We will take a step-by-step approach and create a simple application that increments and decrements a counter to demonstrate the implementation of Redux in React. The main objective is to help developers understand how to use Redux for state management in complex applications within the React

efficiently.

Before integrating Redux, make sure you have the following prerequisites installed:

- **Node.js and npm:** Redux and React.js, both being JavaScript libraries, rely on Node.js for their environment. You can download Node.js, which includes npm (Node Package Manager), from the official Node.js website.
- **React.js:** If you haven't set up a React application, you can quickly create a new project using Create React App. to set up a new project.

Redux is like the brain of our React.js application, and it helps us manage and control our app data and how it behaves. Let's know some important facts about what Redux is and why you should use it in your React app.

What Is Redux?

Redux is like a smart organizer for your React. js-based application. This organizer called the store, remembers everything going on in your app, like what's happening now and what might change.

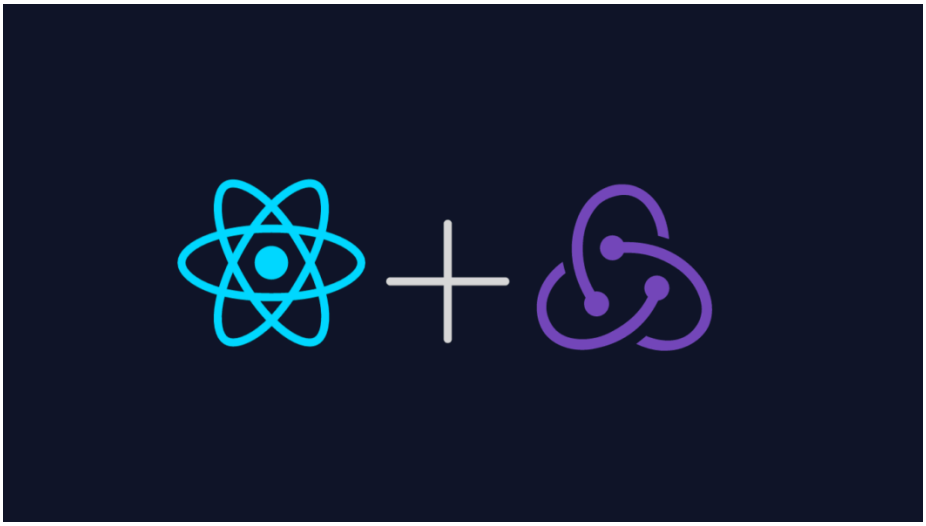
It also has reducers that are like rulebooks. When something in your app changes, these rulebooks say what to do next. They tell your app how to update itself when you make changes.

Why Do We Need Redux in React.js?

Imagine you're building a big, complex React app with lots of components. As your app grows, it becomes more challenging and difficult to keep track of all the data and how it flows between components. This is where you have to use Redux.

Here are some reasons to

passing data through props between different components, making your code cleaner and more organized. **Predictable State Changes:** With Redux, you can predict how your app's state will change because all state changes are controlled by reducers. This predictability makes it easier to debug and understand your app's behavior. **Easy Collaboration**: If you're working on a team, Redux can be a lifesaver. It provides a clear structure for managing the state, making it easier for team members to understand and work on different parts of the app. **Time Travel Debugging**: Redux offers a unique feature called "time travel debugging." It allows you to replay and inspect every action that has ever occurred in your app. This can be incredibly helpful for tracking down bugs. **Scalability**: As your app grows, managing state with plain React can become unwieldy. Redux scales easily with your app's complexity, making it a solid choice for larger projects.



Now, let's get started with integrating Redux into our React.js application.

1. Install NodeJS

NodeJS is an open-source, cross-platform JavaScript runtime environment required for ReactJS applications. If you're an Ubuntu user, execute the following command in your terminal:

following commands:

```
sudo apt-get install -y nodejs
```

If you are using Windows or macOS operating system, you can easily download it from the following sources.Node.js website.

After installation, check the installed Node version:

```
node -v
```

You should get the following output:

```
v18.12.1
```

Automatically with this installation, the system will install an NPM. NPM is a package manager for Javascript programming. To check the installed NPM version, execute the following command:

```
npm -v
```

You should get the following output:

```
8.19.2
```

To update NPM to the latest version available, execute the following command:

```
npm install npm@latest -g
```

You should get the following output:

```
10.2.4
```

2. Install ReactJS and create an application

After installation, check the installed version:

```
create-react-app --version
```

You should get the following output:

```
5.0.1
```

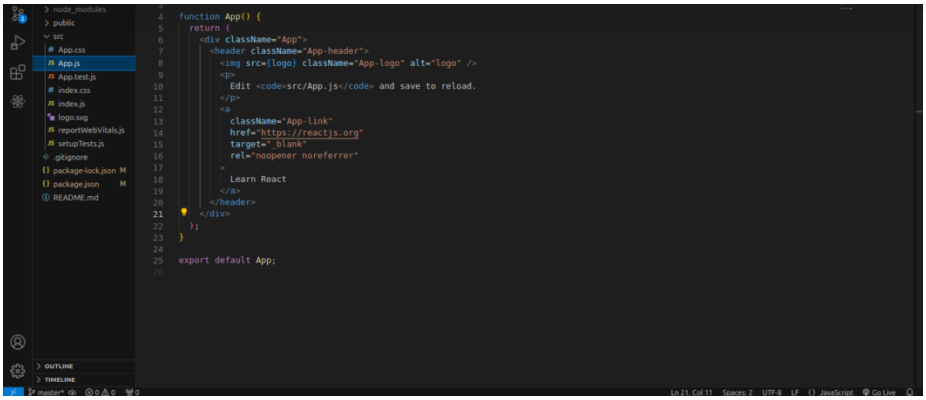
To create the ReactJS app, run the command below. Here, we'll name the app `redux_tutorial`, but you can choose any name.

```
npx create-react-app redux_tutorial
```

You should get the following output:

```
Success! Created redux_tutorial at /home/prasandeep/Desktop/redux_tutorial
Inside that directory, you can run several commands:
  npm start
    Starts the development server.

  npm run build
    Bundles the app into static files for production.

  npm test
    Starts the test runner.

  npm run eject
    Removes this tool and copies build dependencies, configuration files
    and scripts into the app directory. If you do this, you can't go back

We suggest that you begin by typing:

  cd redux_tutorial
  npm start

Happy hacking!
```

Open the created React application folder (e.g, `redux_tutorial`) in your preferred editor, such as VS Code.



3. Installing Redux and Redux Toolkit

Using npm

To use Redux and Redux Toolkit, install dependencies in your React app. Open terminal in created React app directory and run these commands:

```
npm install redux react-redux
npm install @reduxjs/toolkit
```

The first command installs the Redux core and React Redux packages. The second command installs Redux Toolkit separately. This approach offers better control over the installation process, allowing you to manage each package individually.

Using Yarn

For Yarn users, the equivalent commands are:

```
yarn add redux react-redux
yarn add @reduxjs/toolkit
```

Redux serves as the core library, providing fundamental state

To verify if these dependencies were installed, navigate to your package.json file. The dependencies section should list their names

```
{
  "name": "redux_tutorial",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "@reduxjs/toolkit": "^1.9.7",
    "@testing-library/jest-dom": "^5.17.0",
    "@testing-library/react": "^13.4.0",
    "@testing-library/user-event": "^13.5.0",
    "react": "^18.2.0",
    "react-dom": "^18.2.0",
    "react-redux": "^8.1.3",
    "react-scripts": "5.0.1",
    "redux": "^4.2.1",
    "web-vitals": "^2.1.4"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "eslintConfig": {
    "extends": [
      "react-app",
      "react-app/jest"
    ]
  },
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  }
}
```

In the `package.json` file, you can confirm that Redux and Redux Toolkit dependencies have been successfully installed in your React application.

In ReactJs, a Redux store holds our application's state. To create a Redux Store, we need to follow these steps:

To specify how the application's state changes in response to actions, reducers are used. You need to create a new folder inside the `src` folder, named `reducers`. Inside this `reducers` folder, you need to create a new file, for example `index.js`, and implement the reducers state provided below. Here's a basic example:

```
// src/reducers/index.js
const initialState = {
    counter: 0
  };

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

  export default counterReducer;
```

In above code, a `counterReducer` is established to manage a simple counter state. The initial state of the counter is set to zero. The reducer has two actions, `"INCREMENT"` and `"DECREMENT"`, which respectively increase or decrease the counter value. If the action type is not recognized, the current state of the counter is returned.

Combine Reducers (if needed)

If your application has multiple reducers, you can combine them into one using `combineReducers`:

```
// src/reducers/index.js
import { combineReducers } from 'redux';

// Reducers
const initialState = {
```



```
      return { ...state, counter: state.counter + 1 };
    case 'DECREMENT':
```

```
  }
};

// Combine Reducers
const rootReducer = combineReducers({
  counter: counterReducer,
  // Add other reducers here if you have more
});

export default rootReducer;
```

In Redux, `combineReducers` is used to combine various reducers together. This creates a `rootReducer` that manages different parts of the application's state. Each reducer is assigned to a specific key in the store, like the counter. The combined rootReducer controls the overall state of the Redux store.

Create a Redux Store

Navigate `src` folder inside your React app and create a folder name(e.g store), and inside `store` folder create a new file (e.g `store.js`), to configure Redux store:

```
// src/store/store.js
import { configureStore } from '@reduxjs/toolkit';
import counterReducer from '../reducers/index';

const store = configureStore({
  reducer: counterReducer,
});

export default store;
```

This code sets up a Redux store using the `@reduxjs/toolkit` package. It imports two things from `../reducers/index`: the `configureStore` function and a reducer. Using `configureStore`, it creates a store with `counterReducer` as the main reducer, and then exports it for use in the app.

Now we successfully set up a redux store and reducers, it's time to connect it with our React components.

To proceed, navigate to your 'src' folder and create a new subfolder named `components`. Inside this directory, create a new file named `Counter.js`. write the provided code into this file.

To connect our React components to the Redux store, we must use the `connect` function from the `react-redux` library. Here is an example implementation using a simple counter component:

```
//src/components/Counter.js
import React from "react";
import { connect } from "react-redux";

const Counter = ({ counter, increment, decrement }) => {
  return (
    <div>
      <p className="counter_title">Counter: {counter}</p>
      <button className="button" onClick={increment}>
        Increment
      </button>
      <button className="button" onClick={decrement}>
        Decrement
      </button>
    </div>
  );
};

const mapStateToProps = (state) => ({
  counter: state.counter
  //  Use 'counter: state.counter.counter' and replace the above line if y
});

const mapDispatchToProps = (dispatch) => ({
  increment: () => dispatch({ type: "INCREMENT" }),
  decrement: () => dispatch({ type: "DECREMENT" })
});

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

This code creates a component called `Counter` in our React app. It shows a counter value and buttons to increase or decrease it.

- The `mapStateToProps` function connects the component to the

Finally, by using `connect` from `react-redux`, the `Counter` component gets linked to the Redux store, allowing it to both read and change the counter value in the app's state.

Using Redux in Your Components

In this step, you need to navigate to the `src` folder, locate the `App.js` file, and replace the existing code with the provided code. To use Redux globally in our React application, we must import the `Counter` component from the `components` folder.

```
// src/App.js
import React from "react";
import Counter from "./components/Counter";
import "./App.css";

function App() {
  return (
    <div className="App">
      <Counter />
    </div>
  );
}

export default App;
```

In a React application, the main structure is defined in the `App.js` file through the App component. This component renders the Counter component, which handles the display of a counter and buttons to adjust its value. The App component is a basic container that houses the Counter component within a div, and applies a CSS class named "App" for styling purposes. Finally, this component is exported as the primary entry point for the entire application.

Provider Component

To connect our React component `<App />` with the react-redux

```
import { createRoot } from 'react-dom/client';
import { Provider } from 'react-redux';
import store from './store/store';
import App from './App';

const root = document.getElementById('root');

const rootInstance = createRoot(root);
rootInstance.render(
  <Provider store={store}>
    <App />
  </Provider>
);
```

The following code initializes a React application by importing important modules including React, `createRoot` for rendering, `Provider` for Redux, Redux `store`, and the main `App` component. It then locates the HTML element with the 'root' ID, creates a rendering instance using `createRoot`, and displays the entire application there.

The `App` component, wrapped in `Provider`, connects to the Redux store, providing access to its state for all child components, and ultimately renders inside the 'root' HTML element.

6. Run Your Application

Once you've finished coding and linked all components together, it's time to run your app. Just open your terminal or command prompt in the same folder where your project is located. Then, start your app by following command:

Using npm

```
npm start
```

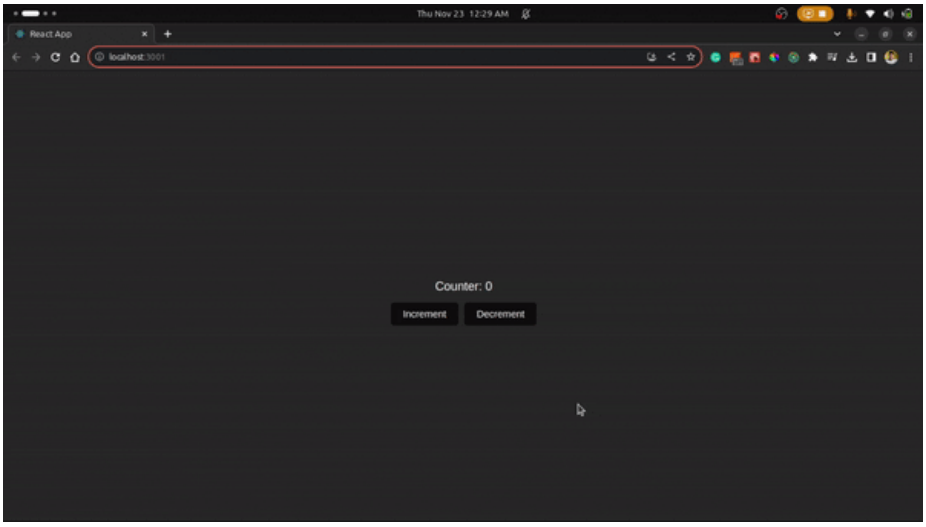
Using Yarn

following output:

```
Compiled successfully!

You can now view redux_tutorial in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://192.168.0.121:3000

Note that the development build is not optimized.
To create a production build, use npm run build.

webpack compiled successfully
```

Now, open your web browser and navigate to http://localhost:3000 (or a different port if specified) to see your Redux-integrated React application in action.

7. Final Output



Link to the GitHub Repository

For code reference, you can access the GitHub repository here.

Conclusion

connect React components to the Redux store.

easiest way to build complex and scalable React applications.

Leave a Reply

Your email address will not be published.

Required fields are marked *

Comment *

Name *

Email *

Post Comment

Writen by:

Prasandeep

I am Prasandeep, CSE Graduate, Full Stack Engineer and Technical Writer

Dan Ackerson

I picked up most of my soft/hardware troubleshooting skills in the US Army. A decade of Java development drove me to operations, scaling infrastructure to cope with the thundering herd. Engineering coach and CTO of Teleclinic.

Blog

Product

Features

Pricing

Docs

Customers

Premium Support

System Status

Security

Terms of Service Privacy Policy