
Embrace Opportunity

Software Engineering Day 12

Redux

Today's Overview

- Introduction to Redux for state management in ReactJS.
- Exercise - Implement redux, identify which state slice should be added to global state in your project and update the code to handle that slice of state in Redux.

- Cheat Sheet:
  - iX Cheat Sheet
  - iX Cheat Sheet - Day 12

Redux

Global State management in React

Redux - Introduction

JavaScript single-page applications have become increasingly complicated, our code must manage more state than ever before. This state can include server responses and cached data, as well as locally created data that has not yet been persisted to the server. UI state is also increasing in complexity, as we need to manage active routes, selected tabs, spinners, pagination controls, and so on. Managing this ever-changing state is hard. Enter Redux.

Redux is a JavaScript library for predictable and maintainable global state management. It helps you write applications that behave consistently, run in different environments (client, server, and native), and are easy to test.

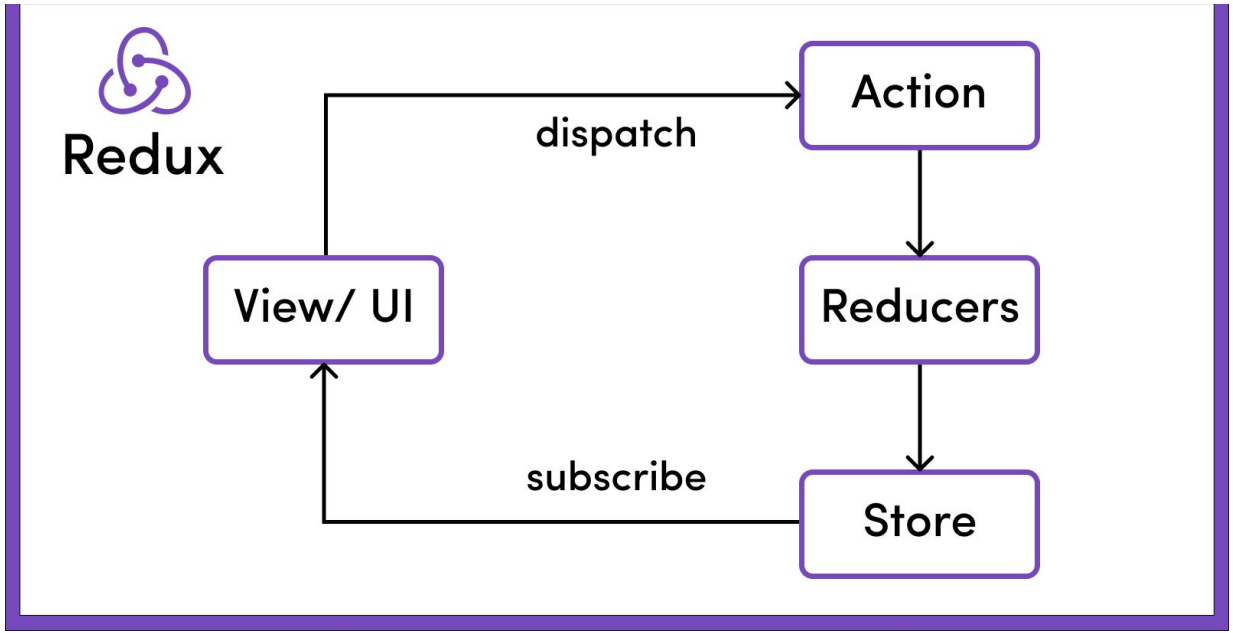
https://redux.js.org/

Redux - Introduction

Redux can be described in three fundamental principles:

- **Single source of truth**
  - The global state of your application is stored in an object tree within a single store.
- **State is read-only**
  - The only way to change the state is to emit an action, an object describing what happened, which executes a reducer.
- **Changes are made with pure functions (reducers)**
  - To specify how the state tree is transformed by actions, you write pure reducers.

Core Principles of Redux



Redux Store

A Redux Store is an object that holds the application's state or state tree. There should only be a single store in a Redux app, as the composition happens on the reducer level. This mean there is always **one source of truth** for global state.

A single state tree also makes it easier to debug or inspect an application, and enables you to persist your app's state in development, for a faster development cycle.

Redux State

The Redux State (also called the state tree) is the single state value that is managed by the store. It represents the entire state of a Redux application, which is often a deeply nested object.

State is read-only and can only be updated by emitting an action which executes a pure reducer to calculate the new state and update the store with the new state vlue.

Redux Action

A Redux Action is a plain object that represents an intention to change the state. Actions are the only way to get data into the store. Any data, whether from UI events, network callbacks, or other sources such as WebSockets needs to eventually be **dispatched** as actions.

Emitting an actions is the only way to change global state. This ensures that neither the views nor the network callbacks will ever write directly to the state. Instead, they express an intent to transform the state. Because all changes are centralized and happen one by one in a strict order, there are no subtle race conditions to watch out for.

Redux Reducer

A Redux Reducer is a function that accepts an accumulation and a value and returns a new accumulation. They are used to reduce a collection of values down to a single value.

In Redux, the accumulated value is the state object, and the values being accumulated are actions. Reducers calculate a new state given the previous state and an action. They must be pure functions—functions that return the exact same output for given inputs. They should also be free of side-effects. This is what enables exciting features like hot reloading and time travel.

Redux Dispatch Function

Dispatch in Redux is a function that synchronously sends an action to the store's reducer, along with the previous state returned by the store, to calculate a new state. It expects actions to be plain objects ready to be consumed by the reducer. **This is the only way to update the store state.**

You can think of dispatching actions as "triggering an event" in the application. Something happened, and we want the store to know about it. Reducers act like event listeners, and when they hear an action they are interested in, they update the state in response.

Redux Toolkit

Redux toolkit is the official, opinionated, batteries-included toolset for efficient Redux development. It intends to be the standard way to write Redux logic. It was originally created to help address three common concerns about Redux:

- "Configuring a Redux store is too complicated"
- "I have to add a lot of packages to get Redux to do anything useful"
- "Redux requires too much boilerplate code"

Most modern React applications use Redux Toolkit as a standard for writing Redux code.

Install Express

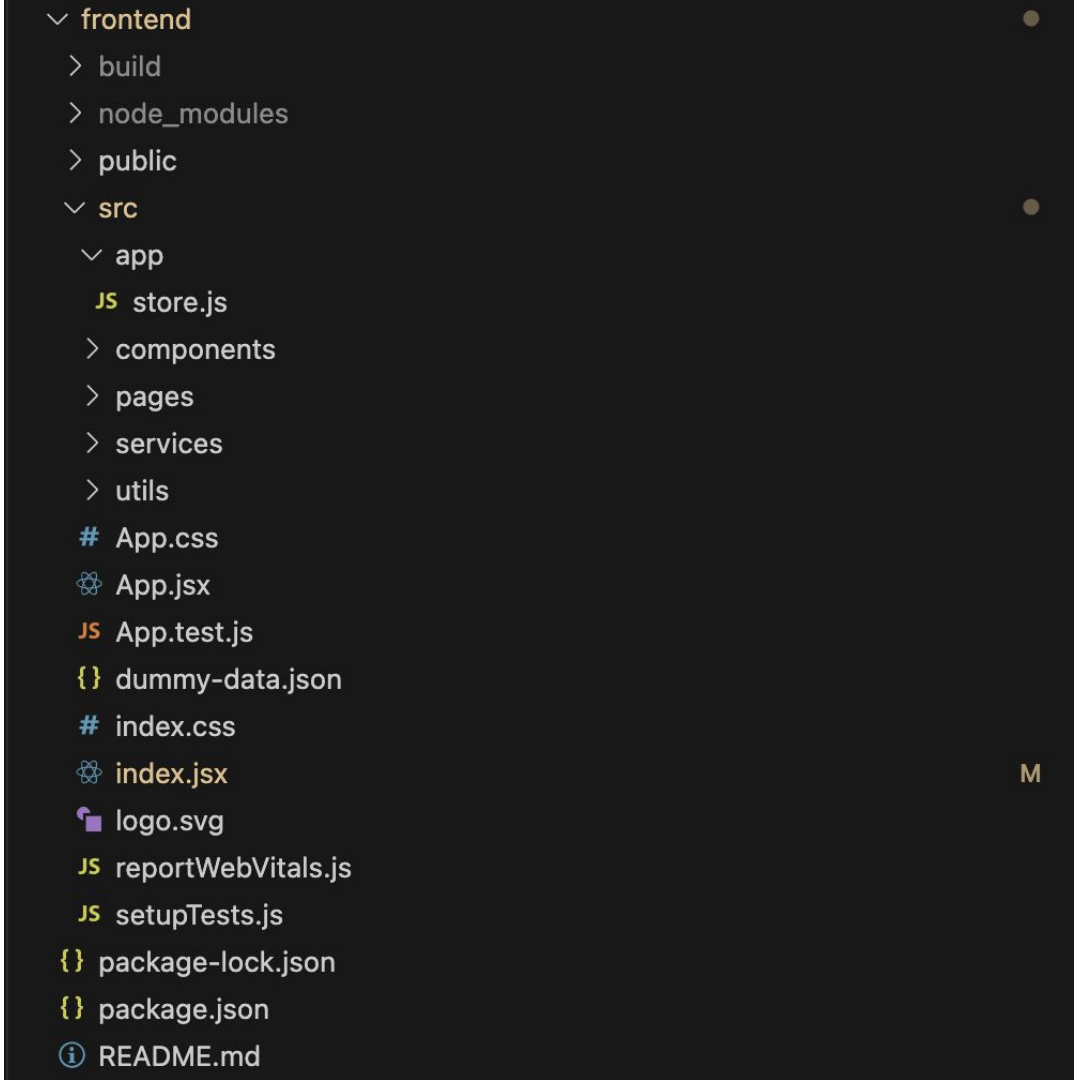
- Run the following command to install Redux and Redux Toolkit.

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

Notice the entry under dependencies in the package.json file.

Create Redux Store

- In *frontend/src/app* create a file called *store.js*



Define Redux Store

- Define Redux store byt importing configureStore from "@reduxjs/toolkit" and instantiate an instance of store with your defined reducers.

```js
import { configureStore } from "@reduxjs/toolkit";
import authReducer from "../features/authSlice";
import blogReducer from "../features/blogsSlice";
import categoryReducer from "../features/categoriesSlice";

export const store = configureStore({
  reducer: {
    auth: authReducer,
    blogs: blogReducer,
    categories: categoryReducer,
  },
});
```

Provide Redux store to react application

- In index.js in the root directory of your project import the Provider element from redux ad import your store created in the last step. Wrap the Base App element in the redux provider element and pass in store as an argument to provide the react application access to the global state and the redux API.

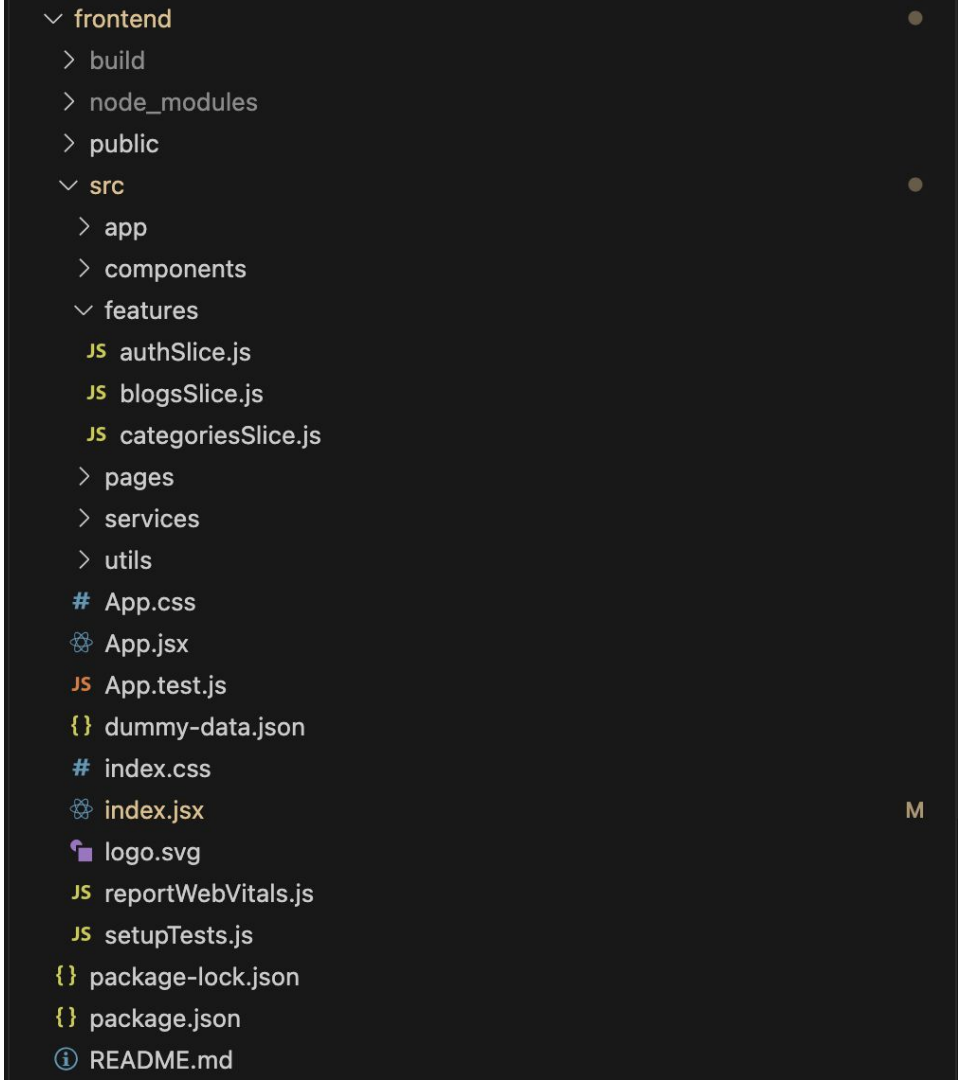
Provide Redux store to react application

```js
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";

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </Provider>
  </React.StrictMode>
);
```

Create Redux Store

- In *frontend/src/features* create a file called *blogSlice.js*



Define Redux Blog State Slice

```js
import { createSlice, createAsyncThunk } from "@reduxjs/toolkit";

import blogService from "../services/blogService";

const initialState = {
  blog: null,
  blogs: [],
  isError: false,
  isSuccess: false,
  isLoading: false,
  message: "",
};

export const blogsSlice = createSlice({
  name: "blogs",
  initialState,
  reducers: {
    reset: (state) => initialState,
  },
});

export const { reset } = blogsSlice.actions;
export default blogsSlice.reducer;
```

Define Redux Reducers

- Since we want to keep the blogs data in state, we need a reducer to fetch the data from the our backend API and set the blogs state. We then need to move the HTTP request from the useEffect hook in the Home page to it's own Reducer in the blog state slice. We can then dispatch the fetchBlogs action to query the API and store the response in global state. We use the createAsyncThunk function from "@reduxjs/toolkit" to handle async reducer functions. This allows the state slice to listen to the events such as pending, fulfilled and rejected.

Define Redux Reducers

We need to then update the useEffect hook in the Home page dispatch the fetchBlogs action to query the API and store the response in global state.

```js
export const fetchBlogs = createAsyncThunk(
  "blogs/fetchBlogs",
  async (_, thunkAPI) => {
    try {
      return await blogService.fetchBlogs();
    } catch (error) {
      const message = error.message || error;
      return thunkAPI.rejectWithValue(message);
    }
  }
);

export const blogsSlice = createSlice({
  name: "blogs",
  initialState,
  reducers: {
    reset: (state) => initialState,
  },
  extraReducers: (builder) => {
    builder
      .addCase(fetchBlogs.pending, (state) => {
        state.isLoading = true;
      })
      .addCase(fetchBlogs.fulfilled, (state, { payload }) => {
        state.blogs = payload.data;
        state.isSuccess = true;
        state.isLoading = false;
        state.message = payload.message;
      })
      .addCase(fetchBlogs.rejected, (state, { payload }) => {
        state.message = payload;
        state.isError = true;
        state.isLoading = false;
      });
  },
});
```

Dispatch actions

```js
import { useDispatch } from "react-redux";
import { fetchBlogs, reset as resetBlogs } from "../../features/blogsSlice";
...

export default function HomePage() {
  ...

  const dispatch = useDispatch();

  useEffect(() => {
    dispatch(fetchBlogs());
    return () => {
      dispatch(resetBlogs());
    };
  }, [dispatch]);

  ...

}
```

We need the Home page component to subscribe the blogs state and update the UI when the state changes. We can do this by importing useSelector from react-redux and destructuring the blog state slice. Every time the global state slice that we are listening to update the component will re render.

```js
import { useDispatch, useSelector } from "react-redux";
import { fetchBlogs, reset as resetBlogs } from "../../features/blogsSlice";
...

export default function HomePage() {
  ...

  const dispatch = useDispatch();
  const {
    blogs,
    isError: isBlogsError,
    isSuccess: blogsSuccess,
    isLoading: isLoadingBlogs,
    message: blogsMessage,
  } = useSelector((state) => state.blogs);

  useEffect(() => {
    dispatch(fetchBlogs());
    return () => {
      dispatch(resetBlogs());
    };
  }, [dispatch]);

  ...

}
```

Exercise

Let’s get stateful

Exercise:

- Create Auth State Slice
  - This includes reducers and actions from
    - Register
    - Login
    - Logout
    - Contain the user state

Homework

Apply what we have learned

Homework

- Update all the page in your application to use Redux state management.
- This includes creating the following reducers:
  - AuthSlice
  - Authorslice
  - BlogSlice
  - CategoriesSlice
- Update the following pages to interact with Redux state
  - Home
  - BlogsPage
  - BlogPage
  - ProfilePage
  - CategoriesPage

Next Class

React Lifecycle Methods, Hooks, Context API and Routes