Redux - A Predictable State Container for JavaScript Apps

Amr A Khllaf

August 3, 2025

Contents

1 What is State Management? 2
1.1 State Management in React . . . 2
1.1.1 1. Local State . . . 2
1.1.2 2. Sharing State Between Components . . . 2
1.1.3 3. Third-Party State Management Libraries . . . 3
1.1.4 4. Custom Hooks . . . 3
1.2 React Query vs Context API . . . 3
1.3 Summary . . . 3

2 Why Choose Redux Over the Context API? 4

3 Redux vs Redux Toolkit 4
3.0.1 Memoization Concept . . . 5

4 Installing Redux Toolkit 5

5 Redux Toolkit Theory 5
5.1 Redux Data Flow . . . 5
5.2 Data Flow: User → View (Component) → dispatch(Action) → Store → Reducer → New State → View updates . . . 6
5.3 Theory Summary . . . 6

6 Configuring Store and Slices 6
6.1 Steps to Create a Redux Store . . . 6

7 Adding Another Slice and Providing the Store 8
7.1 Providing the Store to Your App . . . 8

8 Sharing Data Between Components 9

9 Reducers (Actions) and Dispatching 9

10 Hint about the second Parameters in Reducers 10
10.1 What are the parameters in reducers? . . . 11

11 ExtraReducers and createAsyncThunk 12
11.1 Understanding `createSlice` and `extraReducers` in Redux Toolkit . . . 12
11.1.1 What is `createSlice`? . . . 12

11.1.2 What is `extraReducers`? 12
11.2 Benefits of `createAsyncThunk` 13
11.3 Using `createSlice` with `extraReducers` 13
11.3.1 Summary 14

1 What is State Management?

State management refers to the techniques and patterns used to handle and organize the dynamic data (state) of an application. In web applications, state includes everything from user input and UI controls to API responses and authentication status. As applications grow, efficient state management becomes essential for scalability, maintainability, and predictable behavior.

1.1 State Management in React

React offers several approaches for managing and sharing state between components:

1.1.1 1. Local State

- Managed within individual components using hooks like `useState` and `useReducer`.
- Ideal for simple, isolated state that doesn't need to be accessed elsewhere.

```
function Counter() {
  const [count, setCount] = useState(0);
  return <button onClick={() => setCount(count + 1)}>{count}</button>;
}
```

1.1.2 2. Sharing State Between Components

- **Lifting State Up**: Move state to the closest common ancestor and pass it down as props.
- **Prop Drilling**: Passing state and functions through multiple layers via props, which can become cumbersome in large apps.
- **Context API**: React's built-in Context API allows global state sharing without prop drilling.

```
const ThemeContext = React.createContext();

function App() {
  const [theme, setTheme] = useState("light");
  return (
    <ThemeContext.Provider value={{ theme, setTheme }}>
      <Toolbar />
    </ThemeContext.Provider>
  );
}

function Toolbar() {
  return <ThemeButton />;
}

function ThemeButton() {
  const { theme, setTheme } = useContext(ThemeContext);
```

```
  return (
    <button onClick={() => setTheme(theme === "light" ? "dark" : "light")}>
      Switch Theme ({theme})
    </button>
  );
}
```

This approach allows components to access shared state without passing props through every level.

1.1.3 3. Third-Party State Management Libraries

For complex applications, third-party libraries provide more robust solutions:

- **Redux**: Centralizes application state in a single store, making state changes predictable and traceable.
- **MobX**: Uses observables for reactive state management.
- **Zustand**: Minimal and scalable state management with a simple API.
- **Recoil**: Fine-grained shared state with a focus on performance.
- **Jotai**: Primitive and flexible atomic state management.

1.1.4 4. Custom Hooks

Encapsulate and reuse stateful logic across components by creating custom hooks.

```
function useCounter(initialValue = 0) {
  const [count, setCount] = useState(initialValue);
  const increment = () => setCount((c) => c + 1);
  return { count, increment };
}
```

1.2 React Query vs Context API

- **React Query**: A library for fetching, caching, and synchronizing server state. It simplifies data fetching and provides features like automatic caching, background updates, and query invalidation.
- **Context API**: A built-in React feature for sharing local or global state across components. It does not handle server state or caching.
- **Use Case**: Use React Query for server state (e.g., API data), and Context API for local or global state that doesn't require complex caching or synchronization.

1.3 Summary

- State management is essential for building reliable and maintainable React applications.
- State can be shared by lifting state up, using Context API, or adopting third-party libraries.
- The right approach depends on your application's complexity and scale.
- Tools like Redux, MobX, and Context API help manage and share state efficiently.
- Custom hooks encapsulate reusable stateful logic.
- React Query is ideal for server state; Context API is best for local/global state without complex caching.

---

2 Why Choose Redux Over the Context API?

Redux is often preferred in large-scale applications where state management is complex and demands a robust, predictable solution. Key advantages of Redux over Context API include:

- **Predictable State Updates**: Enforces a strict unidirectional data flow, making state changes explicit and traceable.
- **Centralized Store**: All application state is managed in a single store, simplifying debugging, testing, and persistence.
- **Powerful Middleware**: Supports middleware for handling async logic, logging, error tracking, and more.
- **DevTools Integration**: Redux DevTools offer features like time-travel debugging and action/state inspection.
- **Scalability**: Designed for complex apps, Redux scales well as your app grows.
- **Rich Ecosystem**: Mature ecosystem, extensive documentation, and a large community.
- **Immutability by Design**: Encourages immutable state updates, reducing bugs from unintended mutations.
- **Separation of Concerns**: Separates state management logic from UI components for cleaner, more maintainable code.

While Context API is great for simple or moderately complex state sharing, Redux excels when state logic is intricate, shared across many components, or requires advanced tooling.

---

3 Redux vs Redux Toolkit

Redux is a predictable state container for JavaScript applications, designed to manage state in a consistent and maintainable way using actions, reducers, and a central store. It became a standard for React state management since 2017.

Redux Toolkit is the official, recommended way to write Redux logic. It simplifies Redux setup and usage by providing tools and best practices, including:

- **Simplified Configuration**: `configureStore` sets up the store with sensible defaults.
- **createSlice**: Defines reducers and actions in one place, reducing boilerplate.
- **createAsyncThunk**: Handles async actions, making side effects like API calls easier.
- **Immer Integration**: Allows writing mutable-style code in reducers while maintaining immutability.
- **Built-in DevTools Support**: Automatic integration with Redux DevTools.

Key Differences:

- **Boilerplate**: Redux requires more boilerplate; Redux Toolkit reduces it significantly.
- **Configuration**: Redux Toolkit simplifies store setup with `configureStore`.
- **Best Practices**: Redux Toolkit encourages best practices and patterns.
- **Built-in Features**: Includes utilities like `createSlice` and `createAsyncThunk`.
- **Ease of Use**: Redux Toolkit is more intuitive, especially for newcomers.

- **Community Adoption**: Redux Toolkit is now the standard for Redux usage.
- **Performance**: Optimized for performance, with features like Immer for efficient immutable updates.
- **Memoization**: Built-in support for memoization to optimize performance and prevent unnecessary re-renders.

Summary: Redux Toolkit is a powerful abstraction over Redux that simplifies development, reduces boilerplate, and encourages best practices, making it the preferred choice for most Redux applications.

3.0.1 Memoization Concept

Memoization is an optimization technique that caches function results. When a function is called with the same arguments, it returns the cached result instead of recalculating, improving performance for expensive computations or repeated calls.

In Redux and React, memoization (e.g., with `reselect`) is used to prevent unnecessary re-renders by ensuring components only update when relevant state or props change.

---

4 Installing Redux Toolkit

For React projects, install both Redux Toolkit and React Redux:

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

---

5 Redux Toolkit Theory

5.1 Redux Data Flow

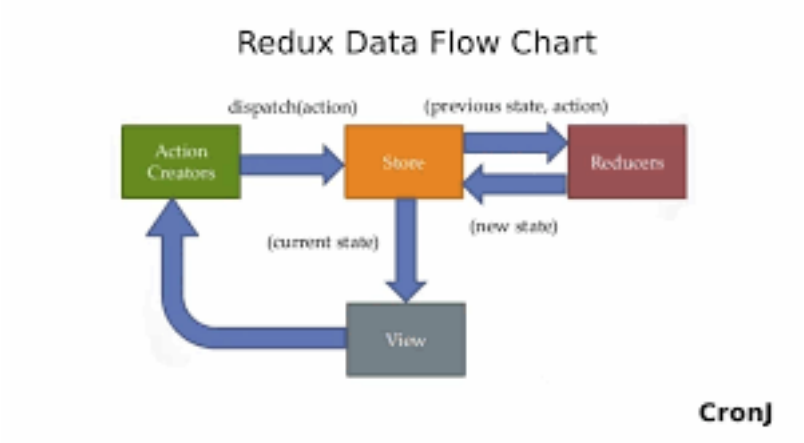

Figure 1: Redux Data Flow Chart

Redux follows a unidirectional data flow:

1. **Action**: A plain object describing a change, with a `type` property.
2. **Dispatch**: The action is dispatched to the Redux store.

3. **Reducers**: Pure functions that take the current state and action, returning a new state.
4. **Store**: Central repository holding the current state.
5. **State**: Immutable object representing application data.
6. **View**: UI updates when state changes.
7. **Selectors**: Functions to extract specific pieces of state.
8. **Components**: React components connect to the store using `useSelector` and `useDispatch`.

5.2 Data Flow: User → View (Component) → dispatch(Action) → Store → Reducer → New State → View updates

1. **User Interaction**: User interacts with the app (e.g., clicks a button).
2. **View (Component)**: React component captures the interaction.
3. **Dispatch Action**: Component calls `dispatch` with an action object.
4. **Store**: Action is sent to the Redux store.
5. **Reducer**: Store passes the action to the reducer, which returns a new state.
6. **New State**: Reducer produces a new state object, stored in the Redux store.
7. **View Updates**: Connected components re-render to reflect the new state.

This flow ensures predictable and traceable state management.

5.3 Theory Summary

- Redux is a predictable state container for JavaScript apps.
- It uses a unidirectional data flow: actions → reducers → store → view.
- Redux Toolkit simplifies Redux with utilities like `configureStore`, `createSlice`, and `createAsyncThunk`.
- The data flow ensures efficient, maintainable, and debuggable state management.

---

6 Configuring Store and Slices

6.1 Steps to Create a Redux Store

1. **Install Redux Toolkit**:

   ```
   npm install @reduxjs/toolkit
   ```

2. **Install React Redux**:

   ```
   npm install react-redux
   ```

3. **Create a `redux` Folder**: In `src`, create a `redux` folder and add `store.js`.

4. **Create the Store** (`store.js`):

```
import { configureStore } from "@reduxjs/toolkit";

configureStore({
```

```
  reducer: {
    // Add your reducers here
  },
});
```

How many stores can you create?

- **Context API**: Multiple stores possible.
- **Redux**: Only one store per app, but you can split state into multiple slices.

5. **Create Slices**: Each slice manages a part of the state.

Example `counterSlice.js`:

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

const counterSlice = createSlice({
  name: "counter",
  initialState: {
    value: 0,
    userName: "Amr",
  },
  reducers: {
    // Reducer functions go here
  },
});

export default counterSlice.reducer;

// Or named export:
const counterReducer = counterSlice.reducer;
export { counterReducer };
```

Importing in the store:

```
import { configureStore } from "@reduxjs/toolkit";
import counterReducer from "./counterSlice";
// or
import { counterReducer } from "./counterSlice";
```

6. **Complete Store Configuration**:

```
import { configureStore } from "@reduxjs/toolkit";
import counterReducer from "./counterSlice";

const store = configureStore({
  reducer: {
    counter: counterReducer,
  },
});
export default store;
```

---

7 Adding Another Slice and Providing the Store

Create `ApiSlice.js`:

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

const apiSlice = createSlice({
  name: "api",
  initialState: {
    allPizza: null,
    isLoading: false,
    isError: false,
    allBeef: null,
  },
});

export default apiSlice.reducer;
```

Update `store.js`:

```
import { configureStore } from "@reduxjs/toolkit";
import counterReducer from "./counterSlice";
import apiReducer from "./ApiSlice";

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

export default store;
```

7.1 Providing the Store to Your App

Wrap your app with the `Provider` from `react-redux`:

```
import React from "react";
import ReactDOM from "react-dom";
import { Provider } from "react-redux";
import store from "./redux/store";

function App() {
  return <Provider store={store}>{/* Your application components
  ↪  */}</Provider>;
}
```

Now, all components can access the Redux state and dispatch actions.

---

8 Sharing Data Between Components

Use the `useSelector` and `useDispatch` hooks from `react-redux`:

- **useSelector**: Selects a piece of state from the Redux store.
- **useDispatch**: Returns the `dispatch` function to dispatch actions.

Example (`About.jsx`):

```
import React from "react";
import { useSelector, useDispatch } from "react-redux";
import { increment, decrement } from "./counterSlice";

function About() {
  const { counter, userName } = useSelector((store) => store.counter);
  const dispatch = useDispatch();

  return (
    <div>
      <h1>About Page</h1>
      <p>Counter: {counter}</p>
      <p>User Name: {userName}</p>
      <button onClick={() => dispatch(increment())}>Increment</button>
      <button onClick={() => dispatch(decrement())}>Decrement</button>
    </div>
  );
}

export default About;
```

---

9 Reducers (Actions) and Dispatching

- **Actions**: Plain objects describing a change, with a `type` property.
- **Reducers**: Pure functions that take the current state and an action, returning a new state.

Reducers in a slice handle state changes for that slice. Example (`counterSlice.js`):

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

const counterSlice = createSlice({
  name: "counter",
  initialState: {
    counter: 0,
    userName: "Amr",
  },
  reducers: {
    increment: (state) => {
      state.counter += 1;
    },
```

```
    decrement: (state) => {
      state.counter -= 1;
      state.userName = "Amr";
    },
    setUserName: (state, action) => {
      state.userName = action.payload;
    },
    changeName: (state) => {
      state.userName = "Omar Khllaf";
    },
  },
});

export const { increment, decrement, setUserName, changeName } =
  counterSlice.actions;
export default counterSlice.reducer;
```

In your component, use `useDispatch` to dispatch actions:

```
import React from "react";
import { useSelector, useDispatch } from "react-redux";
import { increment, decrement } from "./counterSlice";

function About() {
  const { counter, userName } = useSelector((store) => store.counter);
  const dispatch = useDispatch();

  return (
    <div>
      <h1>About Page</h1>
      <p>Counter: {counter}</p>
      <p>User Name: {userName}</p>
      <button onClick={() => dispatch(increment())}>Increment</button>
      <button onClick={() => dispatch(decrement())}>Decrement</button>
    </div>
  );
}

export default About;
```

Note: Always call the action creator inside `dispatch` (e.g., `dispatch(increment())`), as it returns the action object expected by Redux.

---

10 Hint about the second Parameters in Reducers

Note: Reducers property made to include your functions that only can change on slice Initial State and by default each function may take 2 parameters: `state` and `action`.

10.1 What are the parameters in reducers?

- It's and Object that contains two main properties:

1. **State**: The current state of the slice.
2. **Action**: The action object dispatched, containing a `type` and optional `payload`.
   - **Type (Unique Identifier)**: A string that identifies the action, used to determine how the state should change and gives a unique identifier for the action.
   - **Payload**: Optional data passed with the action, used to update the state.

- counterSlice example:

```
import { createSlice } from "@reduxjs/toolkit";
const counterSlice = createSlice({
  name: "counter",
  initialState: {
    counter: 0,
    userName: "Amr",
  },
  reducers: {
    increment: (state, action) => {
      state.counter += action.payload.incrementValue; // Increment the
↪   counter
      alert(`Incremented by ${action.payload.incrementName}`); // Alert
       ↪  with increment name
      state.userName = action.payload.incrementName; // Set userName from
↪   action payload
      // -------------------------------------
      // See the the next code to understand the passing of the payload
      // -------------------------------------
    },
    decrement: (state, action) => {
      state.counter -= action.payload; // Decrement the counter
      state.userName = "Amr"; // Reset userName
    },
    setUserName: (state, action) => {
      state.userName = action.payload; // Set userName from action payload
    },
    changeName: (state) => {
      state.userName = "Omar Khllaf"; // Change userName to a fixed value
    },
  },
});
```

- In This example, the `increment` and `decrement` action creators are used to dispatch actions with payloads.

- About.jsx example:

```
import React from "react";
import { useSelector, useDispatch } from "react-redux";
import { increment, decrement } from "./counterSlice";
function About() {
```

```
  const { counter, userName } = useSelector((store) => store.counter);
  const dispatch = useDispatch();

  return (
    <div>
      <h1>About Page</h1>
      <p>Counter: {counter}</p>
      <p>User Name: {userName}</p>
      <button
        onClick={() =>
          dispatch(increment({ incrementValue: 50, incrementName: "Amr" }))
        }
      >
        Increment
      </button>
      <button onClick={() => dispatch(decrement(10))}>Decrement</button>
    </div>
  );
}
export default About;
```

---

11 ExtraReducers and createAsyncThunk

11.1 Understanding `createSlice` and `extraReducers` in Redux Toolkit

11.1.1 What is `createSlice`?

`createSlice` is a core utility provided by **Redux Toolkit** that streamlines the process of managing state in Redux applications. It allows you to define a "slice" of your application's state, along with the reducers and action creators needed to update that state—all in a single, concise structure.

A typical slice includes:

1. **name**: A unique string identifier for the slice.
2. **initialState**: The default state for this slice.
3. **reducers**: An object containing reducer functions that handle synchronous state updates. Each function automatically generates an action creator.
4. **extraReducers** (optional): A field for handling actions not defined in the `reducers` object, such as those generated by `createAsyncThunk` or actions from other slices.

11.1.2 What is `extraReducers`?

The `extraReducers` property enables your slice to respond to actions that are external to its own `reducers`. This is especially **useful for handling asynchronous actions** created with `createAsyncThunk`, or for reacting to actions dispatched by other slices.

- **Use Case**: When you need to update the slice's state in response to actions not defined in the `reducers` section.

- **How it works**: You can define `extraReducers` as either an object or a builder callback function. The builder pattern is recommended for better TypeScript support and flexibility.

11.2 Benefits of `createAsyncThunk`

When you use `createAsyncThunk`, it automatically generates three action types for handling the lifecycle of an asynchronous request:

1. **pending**: Dispatched when the async operation starts (e.g., when the request is sent and awaiting a response).
2. **fulfilled**: Dispatched when the async operation completes successfully (e.g., when the response is received).
3. **rejected**: Dispatched if the async operation fails (e.g., when an error occurs during the request).

This built-in handling simplifies managing loading, success, and error states in your Redux slices.

11.3 Using `createSlice` with `extraReducers`

When you define a slice using `createSlice`, you can include `extraReducers` to handle actions from `createAsyncThunk` or other slices. This allows your slice to respond to external actions while keeping the state management logic organized and maintainable.

Example Structure

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

// Example async thunk
export const fetchData = createAsyncThunk("data/fetchData", async () => {
  // async logic here
});

const dataSlice = createSlice({
  name: "data",
  initialState: { items: [], status: "idle" },
  reducers: {
    // synchronous reducers here
  },
  extraReducers: (builder) => {
    builder
      .addCase(fetchData.pending, (state) => {
        state.status = "loading";
      })
      .addCase(fetchData.fulfilled, (state, action) => {
        state.status = "succeeded";
        state.items = action.payload;
      })
```

```
      .addCase(fetchData.rejected, (state) => {
        state.status = "failed";
      });
  },
});
```

11.3.1 Summary

- `createSlice` simplifies Redux state management by grouping state, reducers, and actions.
- `extraReducers` lets your slice handle external or asynchronous actions, such as those from `createAsyncThunk`.
- This approach leads to cleaner, more maintainable, and scalable Redux code.

- apiSlice.js File Example:

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

// RTK(Redux Toolkit) => createAsyncThunk
// We will create the functions (outside Slice) that the extraReducers
↪  will use to handle the asynchronous actions, such as fetching data
↪  from an API.
// The Parameters for createAsyncThunk are (type, payloadCreator) which is
↪  a signature of the function that will be called when the action is
↪  dispatched.

// "ecommerce/getAllProducts" : this is mean getAllProducts is refer to
↪  the action type that will be applied on ecommerce slice

export const getAllProducts = createAsyncThunk(
  "ecommerce/getAllProducts",
  function () {
    return fetch("https://ecommerce.routemisr.com/api/v1/products").then(
      (response) => {
        if (!response.ok) {
          throw new Error("Network response was not ok");
        }
        return response.json(); // Parse the JSON response
      }
    );
  }
);

const getAllBrands = createAsyncThunk("ecommerce/getAllBrands", function ()
↪  {
  return fetch("https://ecommerce.routemisr.com/api/v1/brands");
});
// the benefit of createAsyncThunk that it has 3 cases that will be
↪  handled automatically:
// 1. pending: when the request is sent and waiting for the response
```

```
// 2. fulfilled: when the request is successful and the response is
↪  received
// 3. rejected: when the request is failed and the error is received

const forkifySlice = createSlice({
  name: "ecommerce", // Unique name for the slice
  initialState: {
    // Initial state of the slice
    allProducts: null,
    allBrands: null,
    isLoading: false,
    isError: false,
  },
  reducers: {
    // Synchronous reducers can be defined here
  },
  extraReducers: function (builder) {
    // This is where you can handle asynchronous actions such as data
    ↪  fetching, handling API responses, while keeping the state updated,
    ↪  handling loading and error states.
    builder.addCase(getAllProducts.fulfilled, function (state, action) {
      // When the getAllProducts action is fulfilled, update the state
      ↪  with the fetched products
      console.log("Products fetched successfully:", action.payload);
      console.log("Products fetched successfully:", action.payload.data);
      ↪  // Log the data property
      state.allProducts = action.payload.data; // Set the allProducts state
↪  with the fetched data
      state.isLoading = false; // Set loading to false
    });
    builder.addCase(getAllProducts.pending, function (state) {
      // When the getAllProducts action is pending, set loading to true
      state.isLoading = true;
    });
    builder.addCase(getAllProducts.rejected, function (state) {
      // When the getAllProducts action is rejected, set isError to true
      state.isError = true;
      state.isLoading = false; // Set loading to false
    });
  },
});
```

- To use it in About.jsx or any other component, you can import the `getAllProducts` action and dispatch it using `useDispatch`:

```
import React, { useEffect } from "react";
import { useSelector, useDispatch } from "react-redux";
import { getAllProducts } from "./apiSlice";
function About() {
  const dispatch = useDispatch();
  const { allProducts, isLoading, isError } = useSelector(
```

```
    (store) => store.ecommerce // Access the ecommerce slice state
  );

  useEffect(() => {
    // Fetch all products when the component mounts
    dispatch(getAllProducts());
  }, [dispatch]); // Empty dependency array means this effect runs once
↪  when the component mounts

  if (isLoading) return <div>Loading...</div>; // Show loading indicator
  if (isError) return <div>Error loading products.</div>; // Show error
  ↪  message

  return (
    <div>
      <h1>About Page</h1>
      <ul>
        {allProducts &&
          allProducts.map((product) => (
            <li key={product.id}>{product.name}</li>
          ))}
      </ul>
    </div>
  );
}
export default About;
```

- This example demonstrates how to use `createAsyncThunk` to fetch data from an API and handle the loading and error states in a Redux slice. The `extraReducers` section allows you to respond to the lifecycle of the asynchronous action, keeping your state management clean and organized.