Redux Fundamentals, Part 1: Redux Overview

WHAT YOU'LL LEARN

- What Redux is and why you might want to use it
- The basic pieces that make up a Redux app

Introduction

Welcome to the Redux Fundamentals tutorial! **This tutorial will introduce you to the core concepts, principles, and patterns for using Redux**. By the time you finish, you should understand the different pieces that make up a Redux app, how data flows when using Redux, and our standard recommended patterns for building Redux apps.

In Part 1 of this tutorial, we'll briefly look at a minimal example of a working Redux app to see what the pieces are, and in Part 2: Redux Concepts and Data Flow we'll look at those pieces in more detail and how data flows in a Redux application.

Starting in Part 3: State, Actions, and Reducers, we'll use that knowledge to build a small example app that demonstrates how these pieces fit together and talk about how Redux works in practice. After we finish building the working example app "by hand" so that you can see exactly what's happening, we'll talk about some of the standard patterns and abstractions typically used with Redux. Finally, we'll see how these lower-level examples translate into the higher-level patterns that we recommend for actual usage in real applications.

How to Read This Tutorial

This tutorial will teach you "how Redux works", as well as *why* these patterns exist. Fair warning though - learning the concepts is different from putting them into practice in actual apps.

The initial code will be less concise than the way we suggest writing real app code, but writing it out long-hand is the best way to learn. Once you understand how everything fits

together, we'll look at using Redux Toolkit to simplify things. **Redux Toolkit is the recommended way to build production apps with Redux**, and is built on all of the concepts that we will look at throughout this tutorial. Once you understand the core concepts covered here, you'll understand how to use Redux Toolkit more efficiently.

> **INFO**
>
> If you're looking to learn more about how Redux is used to write real-world applications, please see:
>
> - **The "Modern Redux" page in this tutorial**, which shows how to convert the low-level examples from earlier sections into the modern patterns we do recommend for real-world usage
> - **The "Redux Essentials" tutorial**, which teaches "how to use Redux, the right way" for real-world apps, using our latest recommended patterns and practices.

We've tried to keep these explanations beginner-friendly, but we do need to make some assumptions about what you know already so that we can focus on explaining Redux itself. **This tutorial assumes that you know**:

> **PREREQUISITES**
>
> - Familiarity with HTML & CSS.
> - Familiarity with ES6 syntax and features
> - Understanding of the array and object spread operators
> - Knowledge of React terminology: JSX, State, Function Components, Props, and Hooks
> - Knowledge of asynchronous JavaScript and making AJAX requests

If you're not already comfortable with those topics, we encourage you to take some time to become comfortable with them first, and then come back to learn about Redux. We'll be here when you're ready!

Finally, you should make sure that you have the React and Redux DevTools extensions installed in your browser:

- React DevTools Extension:
  - React DevTools Extension for Chrome
  - React DevTools Extension for Firefox

- Redux DevTools Extension:
  - Redux DevTools Extension for Chrome
  - Redux DevTools Extension for Firefox

What is Redux?

It helps to understand what this "Redux" thing is in the first place. What does it do? What problems does it help me solve? Why would I want to use it?

Redux is a pattern and library for managing and updating application state, using events called "actions". It serves as a centralized store for state that needs to be used across your entire application, with rules ensuring that the state can only be updated in a predictable fashion.

Why Should I Use Redux?

Redux helps you manage "global" state - state that is needed across many parts of your application.

The patterns and tools provided by Redux make it easier to understand when, where, why, and how the state in your application is being updated, and how your application logic will behave when those changes occur. Redux guides you towards writing code that is predictable and testable, which helps give you confidence that your application will work as expected.

When Should I Use Redux?

Redux helps you deal with shared state management, but like any tool, it has tradeoffs. There are more concepts to learn, and more code to write. It also adds some indirection to your code, and asks you to follow certain restrictions. It's a trade-off between short term and long term productivity.

Redux is more useful when:

- You have large amounts of application state that are needed in many places in the app
- The app state is updated frequently over time
- The logic to update that state may be complex
- The app has a medium or large-sized codebase, and might be worked on by many people

Not all apps need Redux. Take some time to think about the kind of app you're building, and decide what tools would be best to help solve the problems you're working on.

> **WANT TO KNOW MORE?**
>
> If you're not sure whether Redux is a good choice for your app, these resources give some more guidance:
>
> - **When (and when not) to reach for Redux**
> - **The Tao of Redux, Part 1 - Implementation and Intent**
> - **Redux FAQ: When should I use Redux?**
> - **You Might Not Need Redux**

Redux Libraries and Tools

Redux is a small standalone JS library. However, it is commonly used with several other packages:

React-Redux

Redux can integrate with any UI framework, and is most frequently used with React. **React-Redux** is our official package that lets your React components interact with a Redux store by reading pieces of state and dispatching actions to update the store.

Redux Toolkit

Redux Toolkit is our recommended approach for writing Redux logic. It contains packages and functions that we think are essential for building a Redux app. Redux Toolkit builds in our suggested best practices, simplifies most Redux tasks, prevents common mistakes, and makes it easier to write Redux applications.

Redux DevTools Extension

The **Redux DevTools Extension** shows a history of the changes to the state in your Redux store over time. This allows you to debug your applications effectively, including using powerful techniques like "time-travel debugging".

Redux Basics

Now that you know what Redux is, let's briefly look at the pieces that make up a Redux app and how it works.

> **INFO**
>
> The rest of the description on this page focuses solely on the Redux core library (the `redux` package). We'll talk about the other Redux-related packages as we go through the rest of the tutorial.

The Redux Store

The center of every Redux application is the **store**. A "store" is a container that holds your application's global **state**.

A store is a JavaScript object with a few special functions and abilities that make it different than a plain global object:

- You must never directly modify or change the state that is kept inside the Redux store
- Instead, the only way to cause an update to the state is to create a plain **action** object that describes "something that happened in the application", and then **dispatch** the action to the store to tell it what happened.
- When an action is dispatched, the store runs the root **reducer** function, and lets it calculate the new state based on the old state and the action
- Finally, the store notifies **subscribers** that the state has been updated so the UI can be updated with the new data.

Redux Core Example App

Let's look at a minimal working example of a Redux app - a small counter application:

Because Redux is a standalone JS library with no dependencies, this example is written by only loading a single script tag for the Redux library, and uses basic JS and HTML for the UI. In practice, Redux is normally used by installing the Redux packages from NPM, and the UI is created using a library like React.

> **INFO**
>
> Part 5: UI and React shows how to use Redux and React together.

Let's break this example down into its separate parts to see what's happening.

State, Actions, and Reducers

We start by defining an initial **state** value to describe the application:

```
// Define an initial state value for the app
const initialState = {
  value: 0
}
```

For this app, we're going to track a single number with the current value of our counter.

Redux apps normally have a JS object as the root piece of the state, with other values inside that object.

Then, we define a **reducer** function. The reducer receives two arguments, the current `state` and an `action` object describing what happened. When the Redux app starts up, we don't have any state yet, so we provide the `initialState` as the default value for this reducer:

```
// Create a "reducer" function that determines what the new state
// should be when something happens in the app
function counterReducer(state = initialState, action) {
  // Reducers usually look at the type of action that happened
  // to decide how to update the state
  switch (action.type) {
    case 'counter/incremented':
      return { ...state, value: state.value + 1 }
    case 'counter/decremented':
      return { ...state, value: state.value - 1 }
    default:
      // If the reducer doesn't care about this action type,
      // return the existing state unchanged
      return state
  }
}
```

Action objects always have a `type` field, which is a string you provide that acts as a unique name for the action. The `type` should be a readable name so that anyone who looks at this code understands what it means. In this case, we use the word 'counter' as the first half of our action type, and the second half is a description of "what happened". In this case, our 'counter' was 'incremented', so we write the action type as `'counter/incremented'`.

Based on the type of the action, we either need to return a brand-new object to be the new `state` result, or return the existing `state` object if nothing should change. Note that we update the state *immutably* by copying the existing state and updating the copy, instead of modifying the original object directly.

Store

Now that we have a reducer function, we can create a **store** instance by calling the Redux library `createStore` API.

```
// Create a new Redux store with the `createStore` function,
// and use the `counterReducer` for the update logic
```

```
const store = Redux.createStore(counterReducer)
```

We pass the reducer function to `createStore`, which uses the reducer function to generate the initial state, and to calculate any future updates.

UI

In any application, the user interface will show existing state on screen. When a user does something, the app will update its data and then redraw the UI with those values.

```
// Our "user interface" is some text in a single HTML element
const valueEl = document.getElementById('value')

// Whenever the store state changes, update the UI by
// reading the latest store state and showing new data
function render() {
  const state = store.getState()
  valueEl.innerHTML = state.value.toString()
}

// Update the UI with the initial data
render()
// And subscribe to redraw whenever the data changes in the future
store.subscribe(render)
```

In this small example, we're only using some basic HTML elements as our UI, with a single `<div>` showing the current value.

So, we write a function that knows how to get the latest state from the Redux store using the `store.getState()` method, then takes that value and updates the UI to show it.

The Redux store lets us call `store.subscribe()` and pass a subscriber callback function that will be called every time the store is updated. So, we can pass our `render` function as the subscriber, and know that each time the store updates, we can update the UI with the latest value.

Redux itself is a standalone library that can be used anywhere. This also means that it can be used with any UI layer.

Dispatching Actions

Finally, we need to respond to user input by creating **action** objects that describe what happened, and **dispatching** them to the store. When we call `store.dispatch(action)`, the store runs the reducer, calculates the updated state, and runs the subscribers to update the UI.

```
// Handle user inputs by "dispatching" action objects,
// which should describe "what happened" in the app
document.getElementById('increment').addEventListener('click', function
() {
  store.dispatch({ type: 'counter/incremented' })
})

document.getElementById('decrement').addEventListener('click', function
() {
  store.dispatch({ type: 'counter/decremented' })
})

document
  .getElementById('incrementIfOdd')
  .addEventListener('click', function () {
    // We can write logic to decide what to do based on the state
    if (store.getState().value % 2 !== 0) {
      store.dispatch({ type: 'counter/incremented' })
    }
  })

document
  .getElementById('incrementAsync')
  .addEventListener('click', function () {
    // We can also write async logic that interacts with the store
    setTimeout(function () {
      store.dispatch({ type: 'counter/incremented' })
    }, 1000)
  })
```

Here, we'll dispatch the actions that will make the reducer add 1 or subtract 1 from the current counter value.

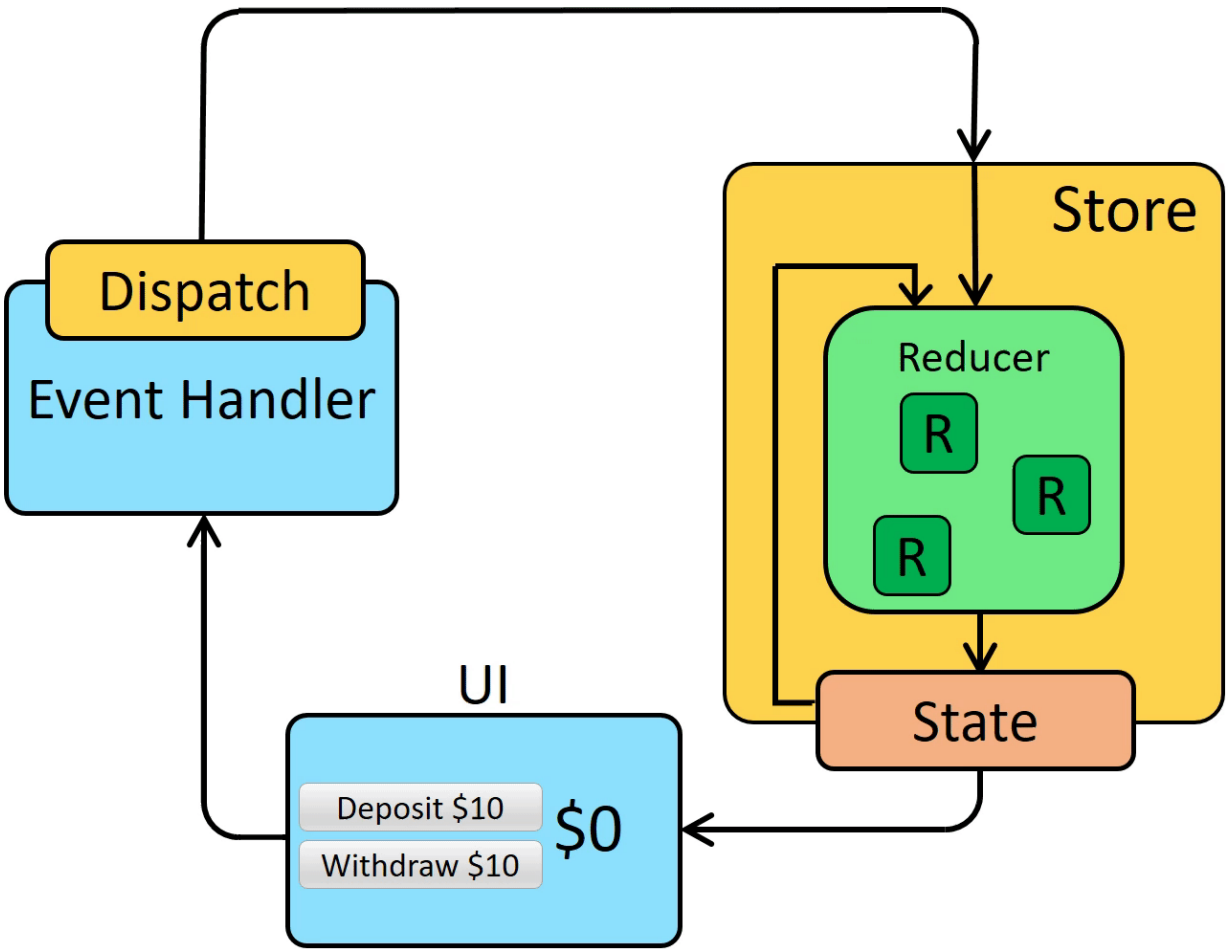
We can also write code that only dispatches an action if a certain condition is true, or write some async code that dispatches an action after a delay.

Data Flow

We can summarize the flow of data through a Redux app with this diagram. It represents how:

- actions are dispatched in response to a user interaction like a click
- the store runs the reducer function to calculate a new state
- the UI reads the new state to display the new values

(Don't worry if these pieces aren't quite clear yet! Keep this picture in your mind as you go through the rest of this tutorial, and you'll see how the pieces fit together.)



What You've Learned

That counter example was small, but it does show all the working pieces of a real Redux app. **Everything we'll talk about in the following sections expands on those basic pieces.**

With that in mind, let's review what we've learned so far:

> **SUMMARY**
>
> - **Redux is a library for managing global application state**

    - Redux is typically used with the React-Redux library for integrating Redux and React together
    - Redux Toolkit is the recommended way to write Redux logic
- **Redux uses several types of code**
    - *Actions* are plain objects with a `type` field, and describe "what happened" in the app
    - *Reducers* are functions that calculate a new state value based on previous state + an action
    - A Redux *store* runs the root reducer whenever an action is *dispatched*

What's Next?

Now that you know what the basic pieces of a Redux app are, step ahead to Part 2: Redux Concepts and Data Flow, where we'll look at how data flows through a Redux app in more detail.