Getting Started with Redux

Redux is a JS 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. On top of that, it provides a great developer experience, such as live code editing combined with a time traveling debugger.

You can use Redux together with React, or with any other view library. It is tiny (2kB, including dependencies), but has a large ecosystem of addons available.

Redux Toolkit is our official recommended approach for writing Redux logic. It wraps around the Redux core, and 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.

RTK includes utilities that help simplify many common use cases, including store setup, creating reducers and writing immutable update logic, and even creating entire "slices" of state at once.

Whether you're a brand new Redux user setting up your first project, or an experienced user who wants to simplify an existing application, **Redux Toolkit** can help you make your Redux code better.

Installation

Redux Toolkit

Redux Toolkit is available as a package on NPM for use with a module bundler or in a Node application:

```
# NPM
npm install @reduxjs/toolkit

# Yarn
yarn add @reduxjs/toolkit
```

Create a React Redux App

The recommended way to start new apps with React and Redux is by using our official Redux+TS template for Vite, or by creating a new Next.js project using Next's `with-redux` template.

Both of these already have Redux Toolkit and React-Redux configured appropriately for that build tool, and come with a small example app that demonstrates how to use several of Redux Toolkit's features.

```
# Vite with our Redux+TS template
# (using the `degit` tool to clone and extract the template)
npx degit reduxjs/redux-templates/packages/vite-template-redux my-app

# Next.js using the `with-redux` template
npx create-next-app --example with-redux my-app
```

We do not currently have official React Native templates, but recommend these templates for standard React Native and for Expo:

- https://github.com/rahsheen/react-native-template-redux-typescript
- https://github.com/rahsheen/expo-template-redux-typescript

Redux Core

The Redux core library is available as a package on NPM for use with a module bundler or in a Node application:

```
# NPM
npm install redux

# Yarn
yarn add redux
```

The package includes a precompiled ESM build that can be used as a `<script type="module">` tag directly in the browser.

For more details, see the Installation page.

Basic Example

The whole global state of your app is stored in an object tree inside a single *store*. The only way to change the state tree is to create an *action*, an object describing what happened, and *dispatch* it to the store. To specify how state gets updated in response to an action, you write pure *reducer* functions that calculate a new state based on the old state and the action.

Redux Toolkit simplifies the process of writing Redux logic and setting up the store. With Redux Toolkit, the basic app logic looks like:

```
import { createSlice, configureStore } from '@reduxjs/toolkit'

const counterSlice = createSlice({
  name: 'counter',
  initialState: {
    value: 0
  },
  reducers: {
    incremented: state => {
      // Redux Toolkit allows us to write "mutating" logic in reducers. It
      // doesn't actually mutate the state because it uses the Immer library,
      // which detects changes to a "draft state" and produces a brand new
      // immutable state based off those changes
      state.value += 1
    },
    decremented: state => {
      state.value -= 1
    }
  }
})

export const { incremented, decremented } = counterSlice.actions

const store = configureStore({
  reducer: counterSlice.reducer
})

// Can still subscribe to the store
store.subscribe(() => console.log(store.getState()))

// Still pass action objects to `dispatch`, but they're created for us
store.dispatch(incremented())
// {value: 1}
store.dispatch(incremented())
// {value: 2}
store.dispatch(decremented())
// {value: 1}
```

Instead of mutating the state directly, you specify the mutations you want to happen with plain objects called *actions*. Then you write a special function called a *reducer* to decide how every action transforms the entire application's state.

In a typical Redux app, there is just a single store with a single root reducer function. As your app grows, you split the root reducer into smaller reducers independently operating on the different parts of the state tree. This is exactly like how there is just one root component in a React app, but it is composed out of many small components.

This architecture might seem like a lot for a counter app, but the beauty of this pattern is how well it scales to large and complex apps. It also enables very powerful developer tools, because it is possible to trace every mutation to the action that caused it. You can record user sessions and reproduce them just by replaying every action.

Redux Toolkit allows us to write shorter logic that's easier to read, while still following the same Redux behavior and data flow.

Legacy Example

For comparison, the original Redux legacy syntax (with no abstractions) looks like this:

```
import { createStore } from 'redux'

/**
 * This is a reducer - a function that takes a current state value and an
 * action object describing "what happened", and returns a new state value.
 * A reducer's function signature is: (state, action) => newState
 *
 * The Redux state should contain only plain JS objects, arrays, and
primitives.
 * The root state value is usually an object. It's important that you should
 * not mutate the state object, but return a new object if the state changes.
 *
 * You can use any conditional logic you want in a reducer. In this example,
 * we use a switch statement, but it's not required.
 */
function counterReducer(state = { value: 0 }, action) {
  switch (action.type) {
    case 'counter/incremented':
      return { value: state.value + 1 }
    case 'counter/decremented':
      return { value: state.value - 1 }
    default:
```

```
      return state
  }
}

// Create a Redux store holding the state of your app.
// Its API is { subscribe, dispatch, getState }.
let store = createStore(counterReducer)

// You can use subscribe() to update the UI in response to state changes.
// Normally you'd use a view binding library (e.g. React Redux) rather than
subscribe() directly.
// There may be additional use cases where it's helpful to subscribe as well.

store.subscribe(() => console.log(store.getState()))

// The only way to mutate the internal state is to dispatch an action.
// The actions can be serialized, logged or stored and later replayed.
store.dispatch({ type: 'counter/incremented' })
// {value: 1}
store.dispatch({ type: 'counter/incremented' })
// {value: 2}
store.dispatch({ type: 'counter/decremented' })
// {value: 1}
```

Learn Redux

We have a variety of resources available to help you learn Redux.

Redux Essentials Tutorial

The **Redux Essentials tutorial** is a "top-down" tutorial that teaches "how to use Redux the right way", using our latest recommended APIs and best practices. We recommend starting there.

Redux Fundamentals Tutorial

The **Redux Fundamentals tutorial** is a "bottom-up" tutorial that teaches "how Redux works" from first principles and without any abstractions, and why standard Redux usage patterns exist.

Learn Modern Redux Livestream

Redux maintainer Mark Erikson appeared on the "Learn with Jason" show to explain how we recommend using Redux today. The show includes a live-coded example app that shows how to

use Redux Toolkit and React-Redux hooks with TypeScript, as well as the new RTK Query data fetching APIs.

See the "Learn Modern Redux" show notes page for a transcript and links to the example app source.

Additional Tutorials

- The Redux repository contains several example projects demonstrating various aspects of how to use Redux. Almost all examples have a corresponding CodeSandbox sandbox. This is an interactive version of the code that you can play with online. See the complete list of examples in the **Examples page**.
- Redux creator Dan Abramov's **free "Getting Started with Redux" video series** and **Building React Applications with Idiomatic Redux** video courses on Egghead.io
- Redux maintainer Mark Erikson's **"Redux Fundamentals" conference talk** and **"Redux Fundamentals" workshop slides**
- Dave Ceddia's post **A Complete React Redux Tutorial for Beginners**

Other Resources

- The **Redux FAQ** answers many common questions about how to use Redux, and the **"Using Redux" docs section** has information on handling derived data, testing, structuring reducer logic, and reducing boilerplate.

- Redux maintainer Mark Erikson's **"Practical Redux" tutorial series** demonstrates real-world intermediate and advanced techniques for working with React and Redux (also available as **an interactive course on Educative.io**).
- The **React/Redux links list** has categorized articles on working with reducers and selectors, managing side effects, Redux architecture and best practices, and more.
- Our community has created thousands of Redux-related libraries, addons, and tools. The **"Ecosystem" docs page** lists our recommendations, and there's a complete listing available in the **Redux addons catalog**.

Help and Discussion

The **#redux channel** of the **Reactiflux Discord community** is our official resource for all questions related to learning and using Redux. Reactiflux is a great place to hang out, ask questions, and learn - come join us!

You can also ask questions on Stack Overflow using the **#redux tag**.

If you have a bug report or need to leave other feedback, please file an issue on the Github repo

Should You Use Redux?

Redux is a valuable tool for organizing your state, but you should also consider whether it's appropriate for your situation. **Don't use Redux just because someone said you should - take some time to understand the potential benefits and tradeoffs of using it**.

Here are some suggestions on when it makes sense to use Redux:

- You have reasonable amounts of data changing over time
- You need a single source of truth for your state
- You find that keeping all your state in a top-level component is no longer sufficient

> **For more thoughts on how Redux is meant to be used, see:**
>
> - **Redux FAQ: When should I use Redux?**
> - **You Might Not Need Redux**
> - **The Tao of Redux, Part 1 - Implementation and Intent**
> - **The Tao of Redux, Part 2 - Practice and Philosophy**
> - **Redux FAQ**

Edit this page

*Last updated on **Mar 31, 2024***